CHEF™

Chef Training Services

Test Driven Cookbook Development

Instructor Guide

Course Rev 1.1.0

1: Introduction

Welcome to Test Driven Cookbook Development.

Slide 2

Introduce Yourselves

Name

Current job role

Previous job roles / Background

Experience with Chef

Favorite Text Editor

Before we start let me introduce myself. Then I would like it if everyone had a chance to introduce themselves.

Instructor Note: Often times with larger, in-person groups I prefer to have the individuals perform this introduction one-on-one. Having people leave their desks and greet as many people as they can during the time allotted. I often feel this works better as it removes the pressure from the single individual to introduce themselves in a way that is presenting themselves and not actually greeting people. When Online, I create a pre-defined order, announce that order, and then invite a person to speak, thank them when they are done.

Slide 3

Expectations

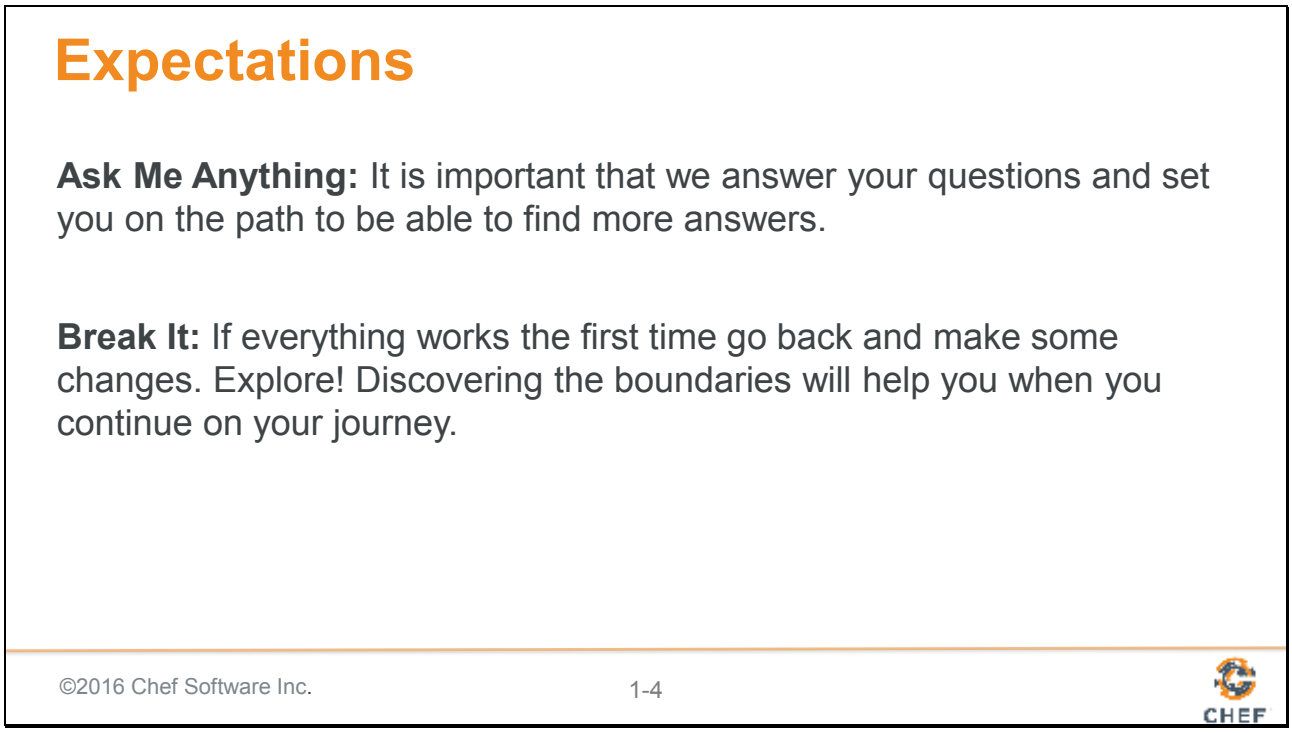
You will leave this class with the confidence to create and extend a cookbook using ChefSpec and InSpec in a test-driven development workflow.

You bring with you your own domain expertise and problems. Chef is a framework for solving those problems. Our job is to teach you how to express solutions to your problems with Chef.

The goal of this class is to give you an introduction to these core testing tools that focuses on the workflow and thought process on why and how to best employ them. Testing is huge domain and will have to keep a tight focus on the challenges and exercises presented in this content. During and throughout the content we will have discussion where we may have additional time to talk about many different topics but in this interest of time and popular opinion we may need to leave those discussions.

During the introductions you learned about the other individuals here in the course with you. They may have shared similar problems and domains. During the time that we are here respectfully reach out them so that you can continue the conversation, grow each other's knowledge, and become better professionals.

Slide 4



All throughout this training I strongly encourage you to ask questions whenever you do not understand a topic, an acronym, concept, or software. By asking a question you better your learning and often times better the learning of those with you in this training. Asking questions is a sign of curiosity that we want to encourage and foster while we are here together.

This curiosity can also be employed by exploring the boundaries of the tools you are using and the language you are writing. The exercises and the labs we will perform will often lead you through examples that work from the beginning to the end. When you develop solutions it is rare that something works from the start all the way to the end. Errors and issues come up from typos or the incorrect usage of a command of the programming language. When you fall off the path it can often be hard to find your way back. Here, if you find yourself always on the correct path explore what happens when you step off of it, what you see, the error messages you are presented with, the new results you might find.

Slide 5

Testing

Testing is a large domain of tools, languages, and practices.

Learning Testing is like learning a language. You will reach fluency fast but it will take practice until you become comfortable.

The best way to improve your testing skill is to write tests.

Testing is a large topic and we will spend the training focused on the core workflow of test-driven development, the tools, and the language constructs. During this time you will be performing a lot of hands on work by executing the commands and write the source code necessary to complete the objectives. While this means we will not move through a larger body of content it will mean that you have a better understanding of the material and have built the important muscle memory to perform this work after this training is done.

Slide 6

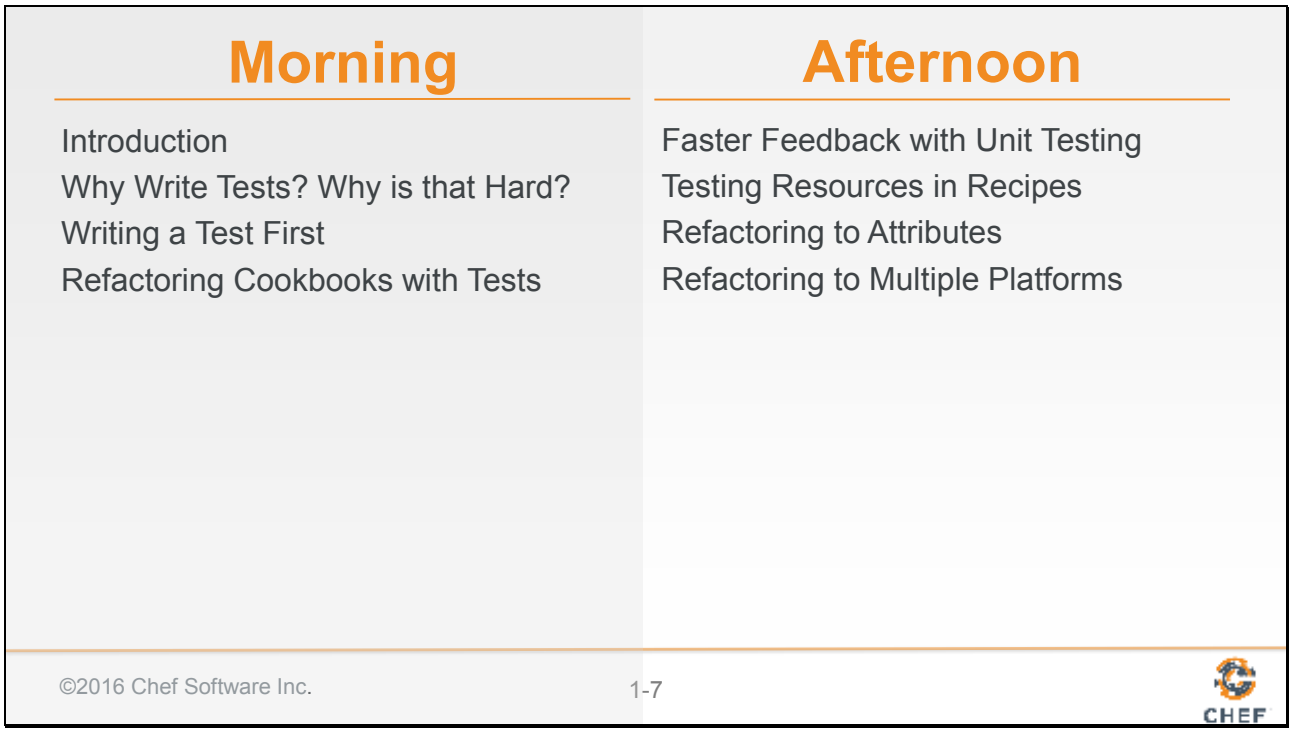
Group Exercises, Labs, and Discussion

This course is designed to be hands on. You will run lots of commands, write lots of code, and express your understanding.

- **Group Exercises:** All participants and the instructor will work through the content together. The instructor will often lead the way and explain things as we proceed.
- **Lab:** You will be asked to perform the task on your own or in groups.
- **Discussion:** As a group we will talk about the concepts introduced and the work that we have completed.

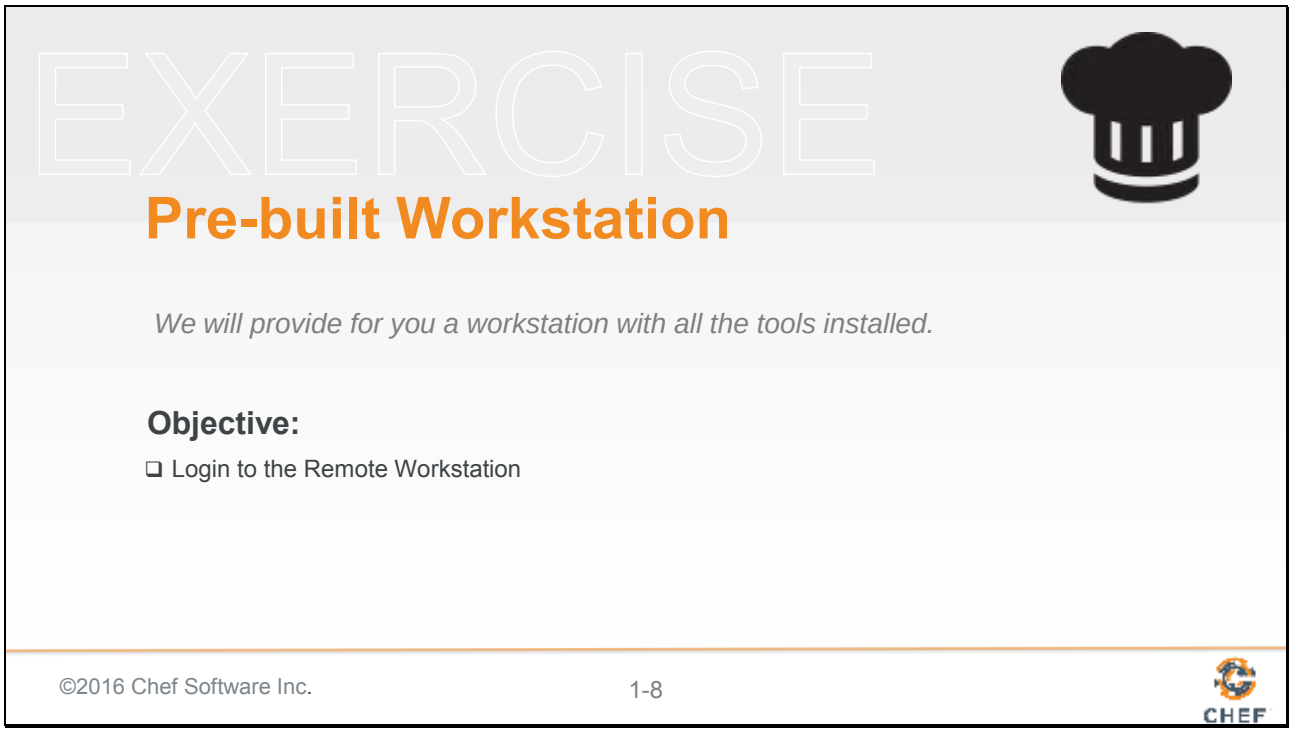
The content of this training has been designed in a way to emphasize this hands-on approach to the content. Together, we will perform exercises together that accomplish an understood objective. After that is done you will often emphasize an activity by performing a lab. The lab is designed to challenge your understanding and retention of the previously accomplished exercises. You can work through this labs on your own or in groups. After completing the labs we will all come together again to review the exercise. Finally, we will end each section with a discussion about the topics that we introduced. These discussions will often ask you to share your opinions, recent experiences, or previous experiences within this domain.

Slide 7



This is the outline of the events for this training. Please take a moment to review this list to ensure that the topics listed here meet your expectations. Take a moment to note which topics are of most interest to you. Also note which topics are not present here on this list. We will discuss your thoughts at the end of the section.

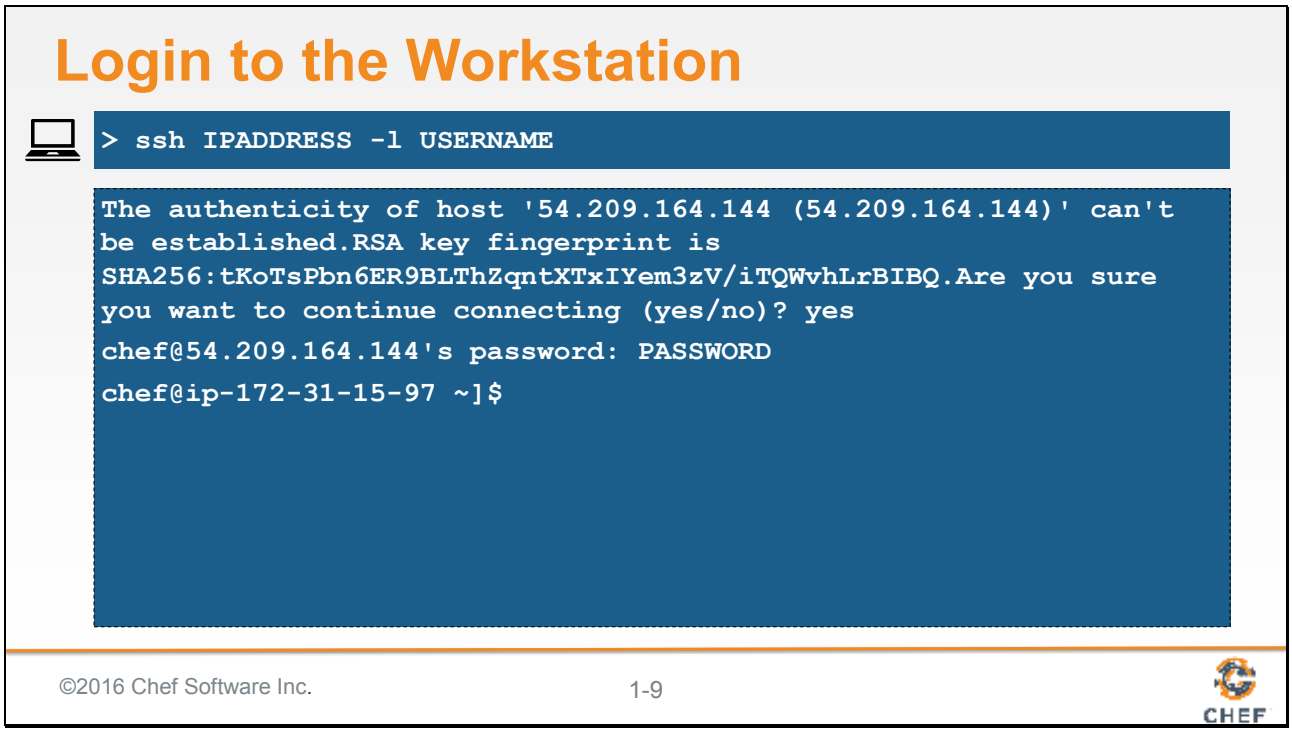
Slide 8



As I mentioned there is a lot work planned for the day. To ensure we focus on the concepts we introduce and not on troubleshooting systems we are providing you a workstation with the necessary tools installed to get started right away.

Instructor Note: At the end of the training it is often a good idea to offer your services to help individuals install necessary software or troubleshoot their systems.

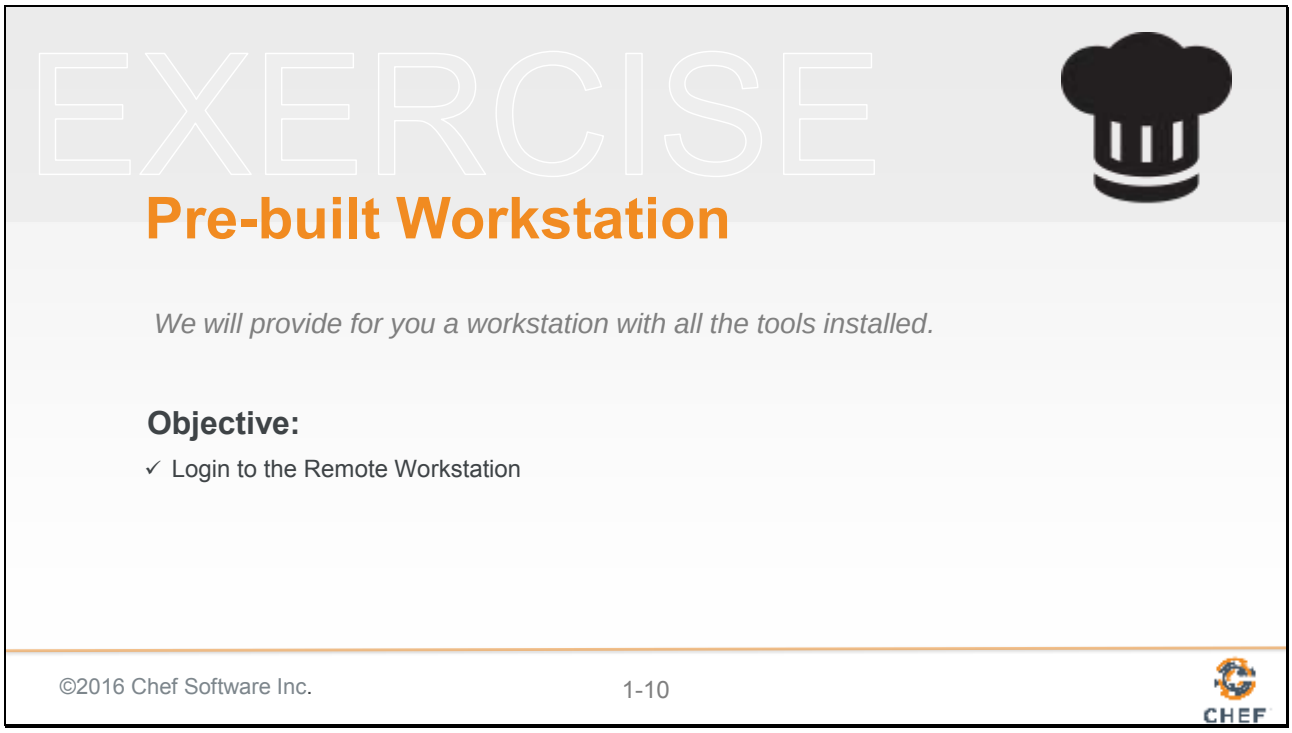
Slide 9



I will provide you with the address, username and password of the workstation. With that information you will need to use the SSH tool that you have installed to connect that workstation.

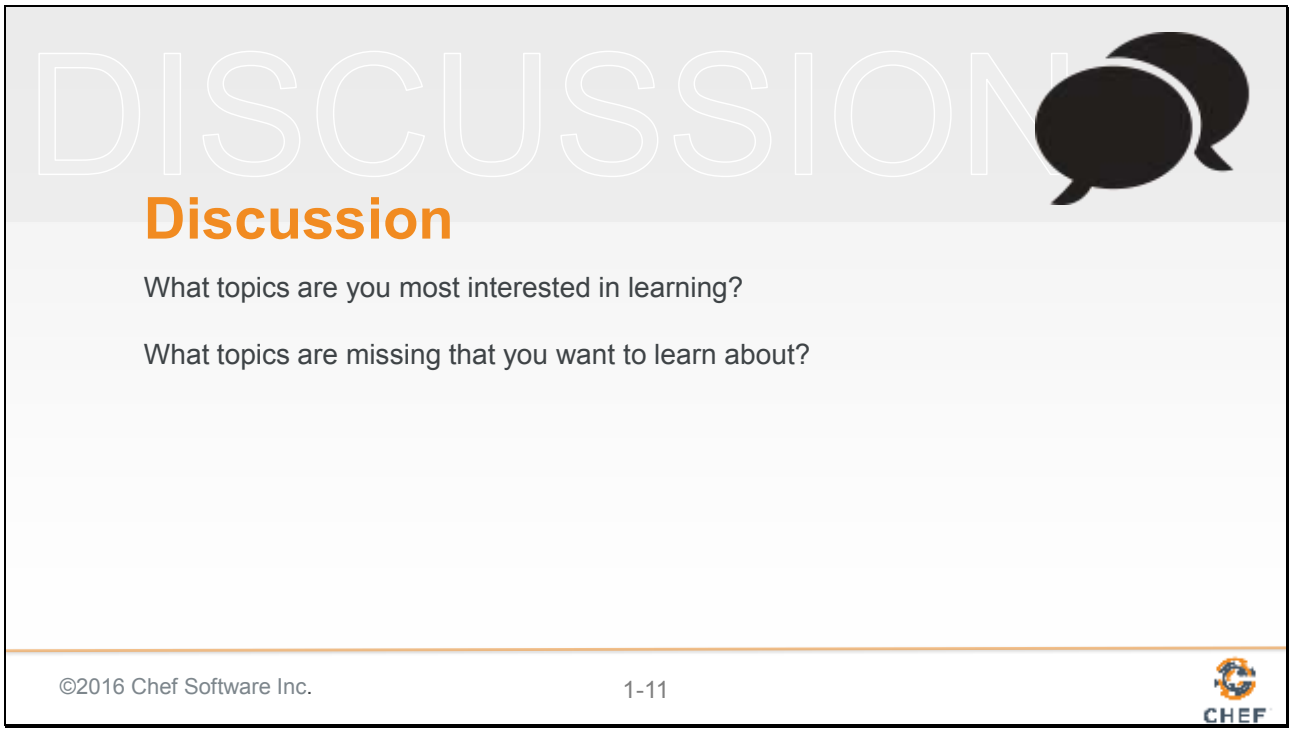
This demonstrates how you might connect to the remote machine using your terminal or command-prompt if you have access to the application ssh. This may be different based on your operating system.

Slide 10



Now that you are connected to that workstation we have taken care of all the necessary work to get started with the training.

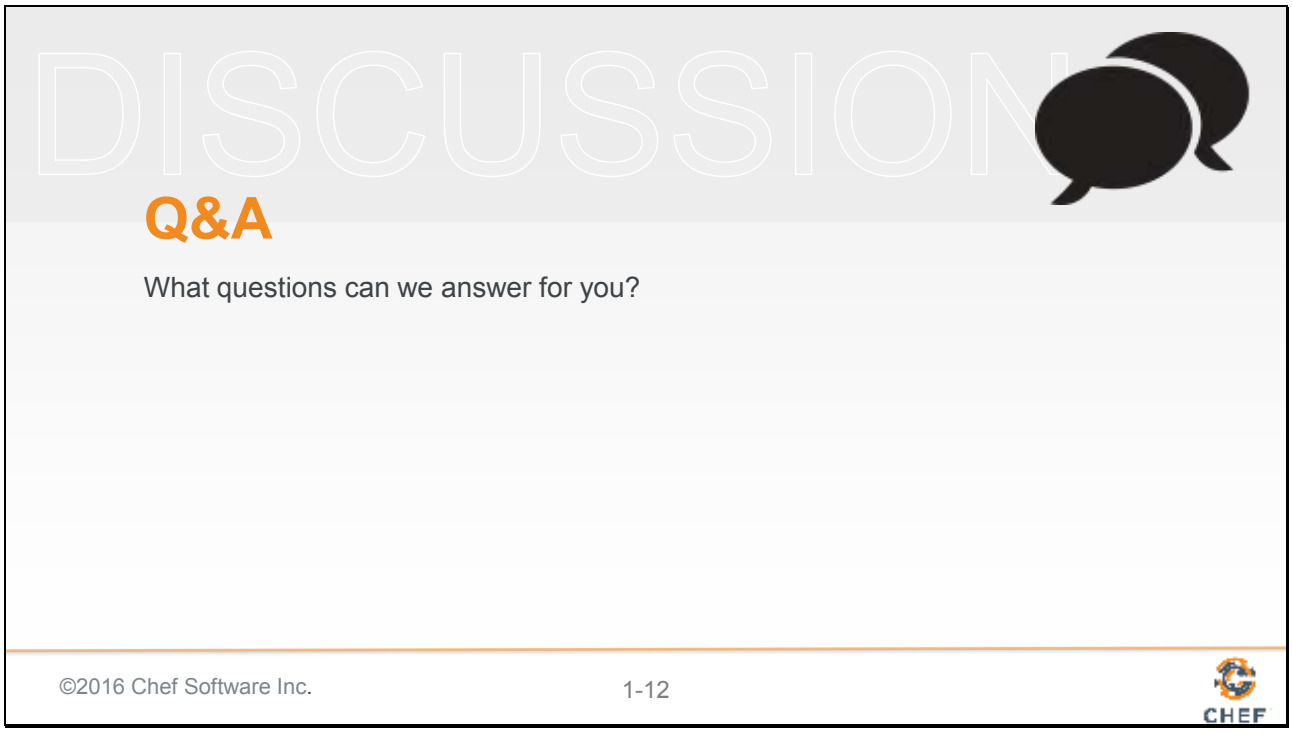
Slide 11



Let us end with a discussion about the following topics.

Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

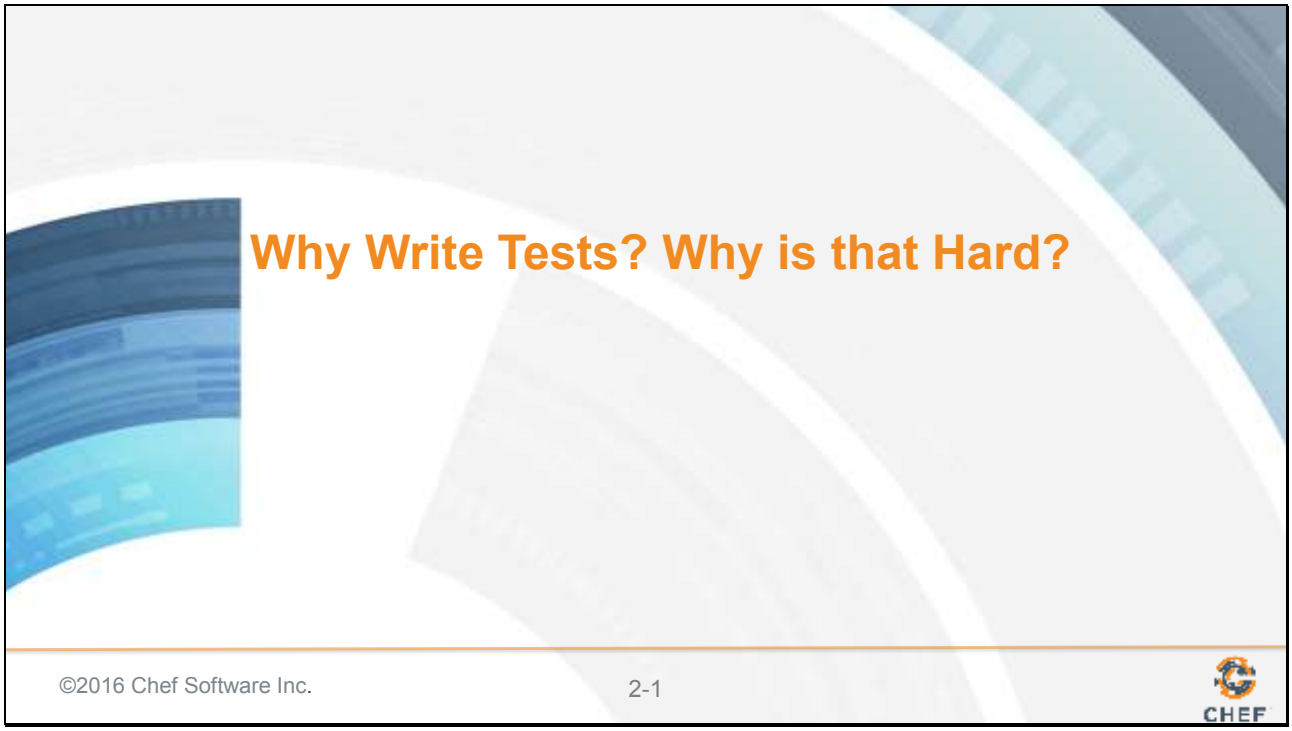
Slide 12



Before we continue let us stop for a moment answer any questions that anyone might have at this time.

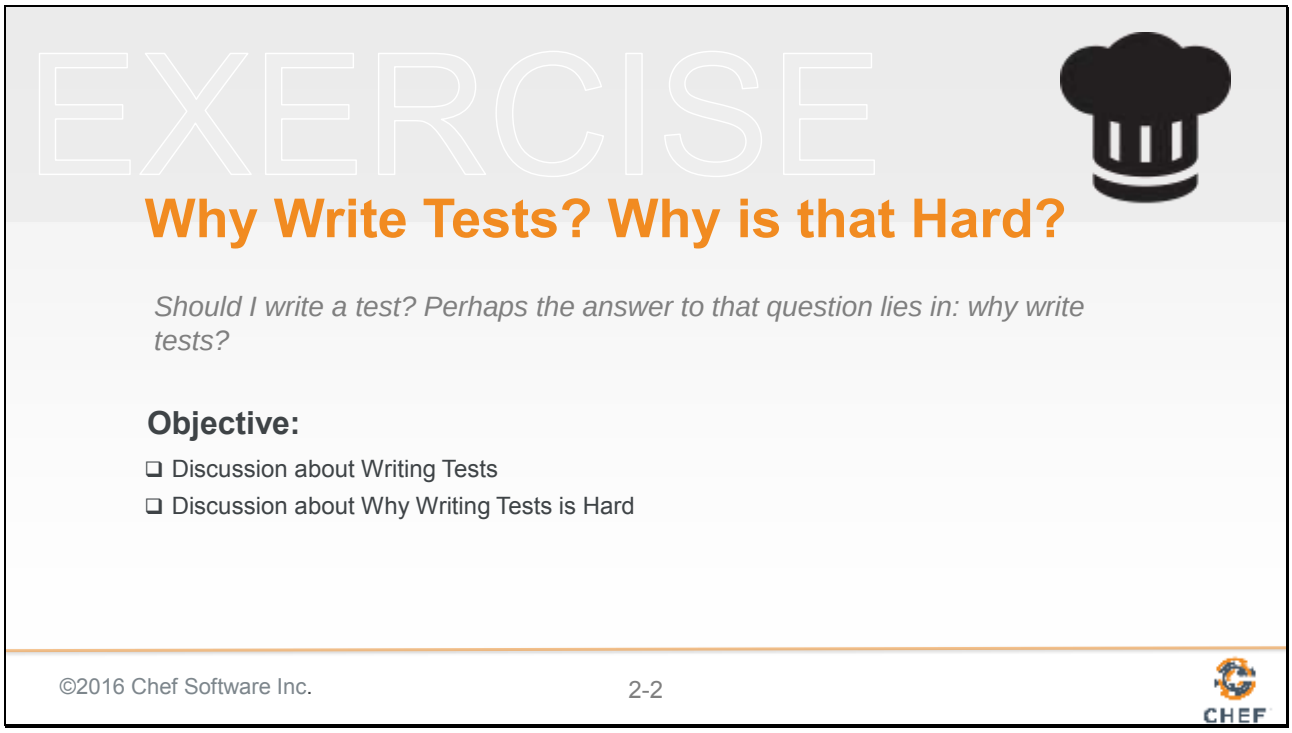
Slide 13

2: Why Write Tests? Why is that Hard?



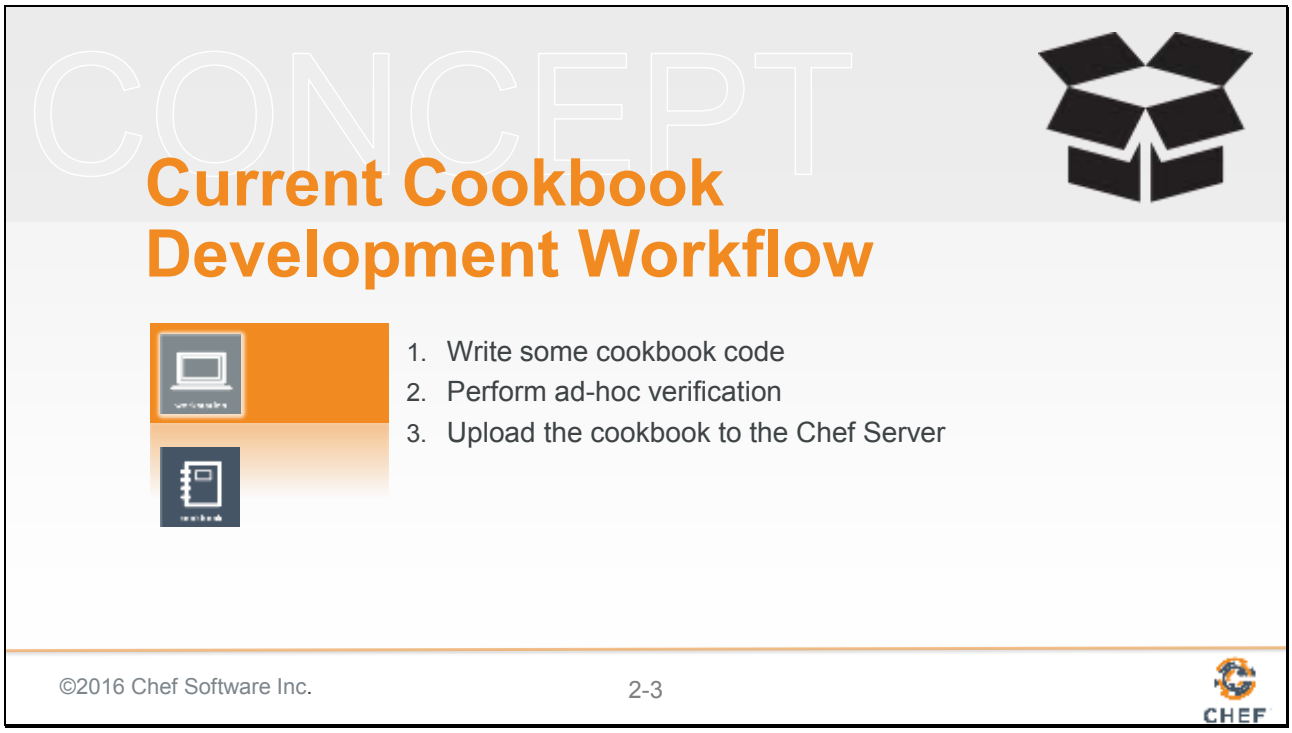
Why should you write tests? Why is important that we write tests for the recipes and the cookbooks that we define. Some of you here may be because you are starting to see an importance to what testing can provide. Others of you may not be convinced. Wherever you stand the real reason you came here to learn is to break down the barriers that make testing hard. Because testing is hard!

Slide 2



All of you likely have a personal answer or opinions to these questions. Good. Capture those because we will have a discussion together. To start the discussion I will provide my thoughts and opinions about why I think it is important to write tests. Then I want you to share your thoughts. Then we will discuss the many reasons that testing is hard.

Slide 3



To understand why it is important to write tests I believe it is important to examine the current cookbook development workflow that most individuals employ.

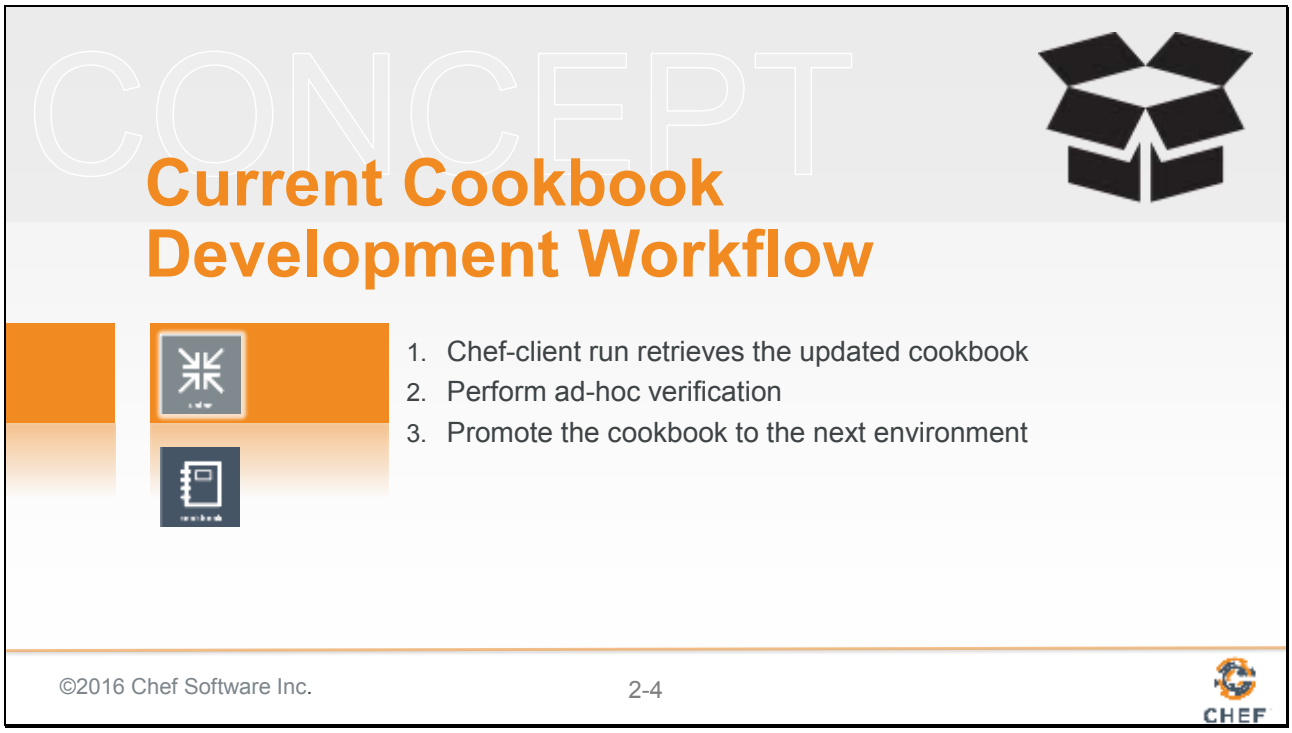
To provide a few answers to why writing tests are powerful and why are they hard to write we need to look at our current cookbook development workflow.

On your local workstation you will write cookbook code. Creating a new recipe to meet new requirements, fixing a bug in an existing recipe, or refactoring complicated recipes into several smaller recipes, helper methods, or maybe even a custom resource.

When you are done with those changes you will spend a few moments visually scanning the code to ensure that your syntax is correct. That every block you start with a 'do' has a matching 'end'. Check your node attributes for spelling issues. Each key-value pair within the hash has a comma that follows.

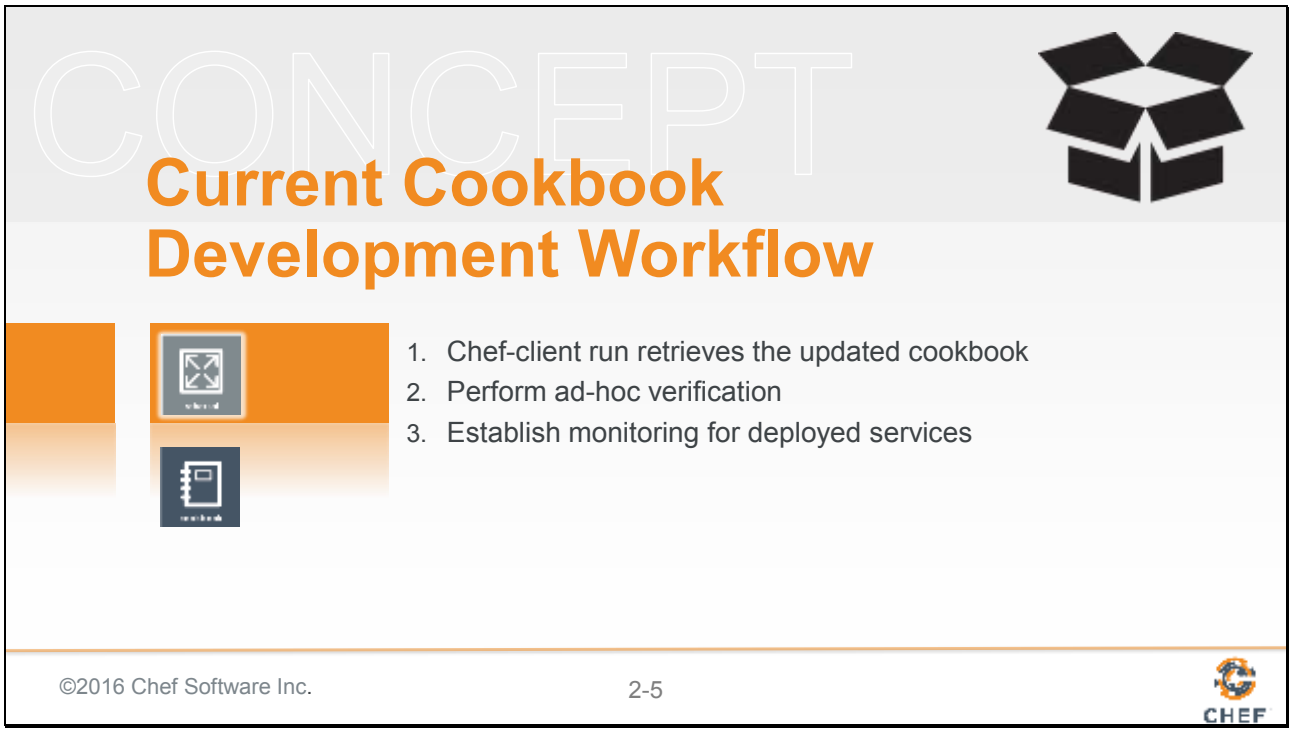
After enough examination we feel comfortable to upload the cookbook to the Chef Server.

Slide 4



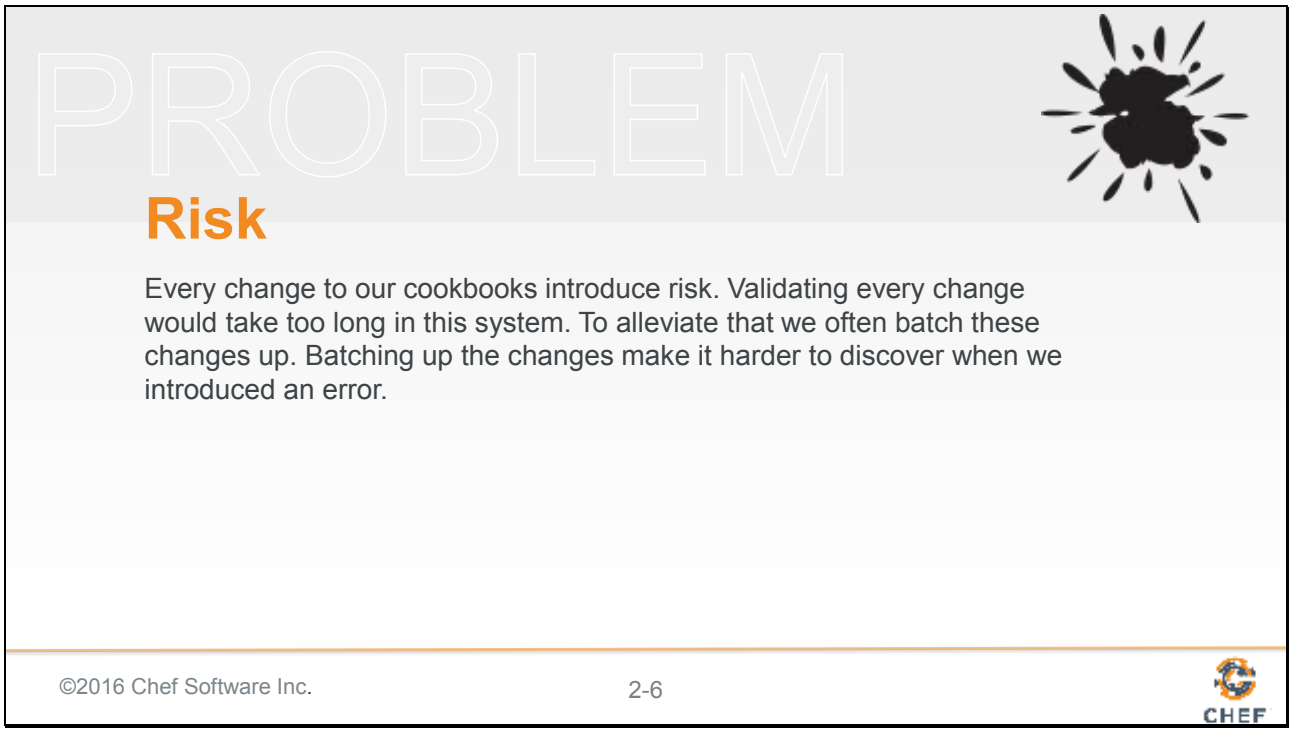
You login to a test node that you patiently bootstrap into a union environment. This is an environment we setup with no cookbook restrictions allowing chef-client to synchronize and apply the latest changes in the recently completed cookbook. Here you see if you got the right package names, spelled all our cookbook attributes correctly, and didn't typo any of the configuration in the templates. If everything converges without error you poke around the system -- running a few commands to see if ports are blocked, services are running, and the logs don't show any errors. Logging out of the working system you feel pretty comfortable promoting the cookbook to the rehearsal environment.

Slide 5



Here in this new environment you may log into another system. Manually perform a chef-client run and then poke around again if everything works. You also may not. It was such a small change and everything worked on the other machine -- so it's likely to work here. Right? Instead of running through a series of ad-hoc verifications again on a new system in this environment - you start to think of the backlog of things that need to get done.

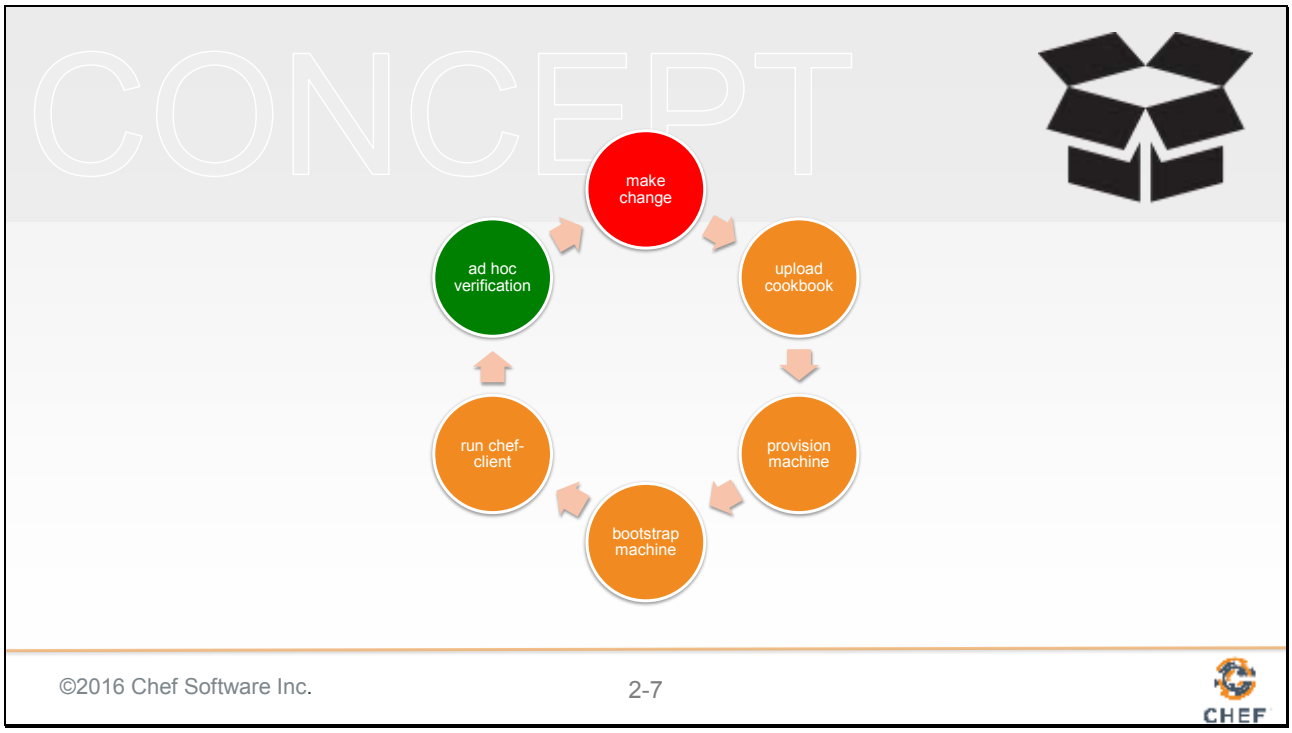
Slide 6



Every time we make changes to our cookbooks we are introducing risk. Ideally we would validate every change to the cookbook but often do not because the amount of time it takes is far too prohibitive. Instead we often will batch up these changes into a set that we will validate. A set of changes like this can often hide errors that we may have introduced. This is definitely true as the complexity of the cookbook code increases.

We have a choice. We can slow down; validating every change. We can also stop making changes altogether. Or we can can adopt new practices, like testing, to help us validate these changes faster; allowing us to continue to move quickly as we continue to satisfy new requirements.

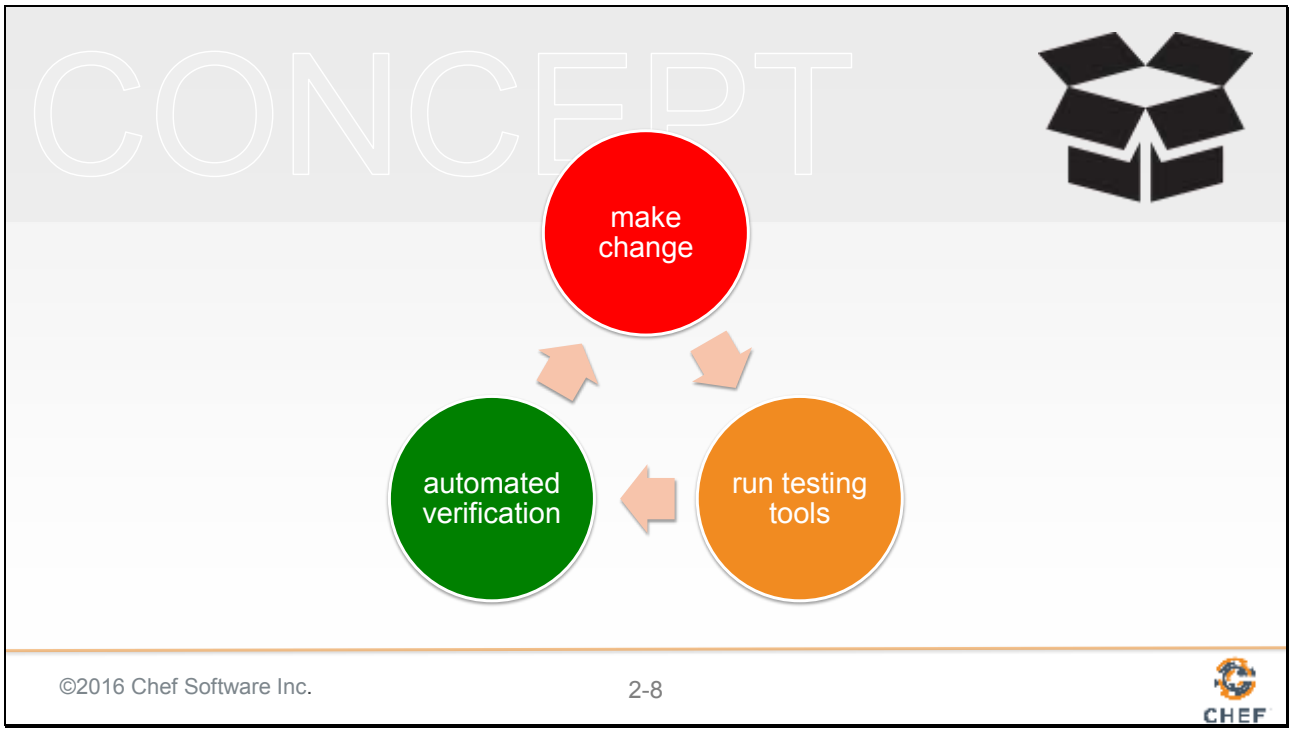
Slide 7



Carrying out testing at every stage (e.g. union, rehearsal) gives great feedback on its success at the cost of the time required for each cookbook to be pushed through this workflow.

Every change needs to be verified in this manner because Ruby, the language Chef is built on, is a dynamically typed programming language. Dynamically typed languages do checking at run-time as opposed to compile-time. This means that ruby files in our cookbook are not executed, thus not validated, until they are run. We also have the problem that we may even write the Ruby correctly but fail to understand the state of the host Operating System (OS) we are attempting to deploy against.

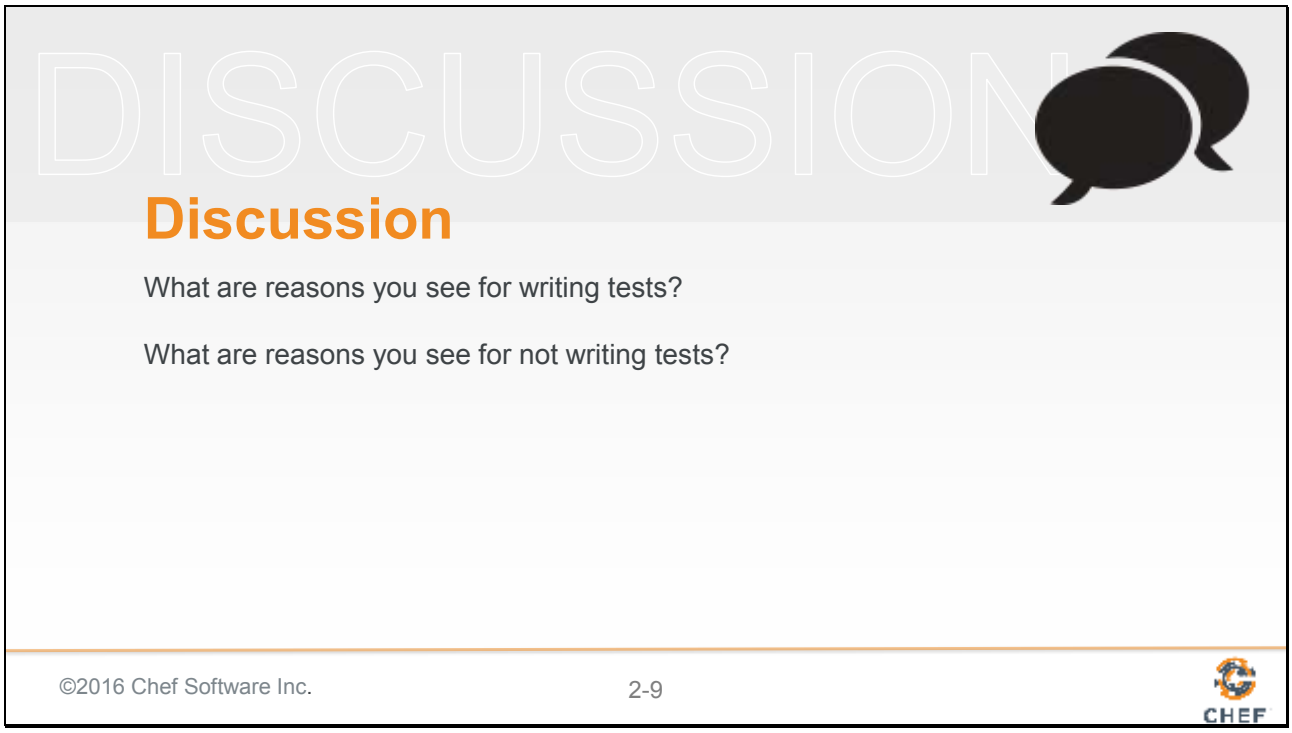
Slide 8



Writing and executing tests decreases the amount of time spent between when you make a change to when you can verify that chance. This reduces the risk within the system.

How testing does address the speed of execution is by removing many of the outside dependencies and allowing you to execute your recipes against in-memory representations of the environment. Or automating the management of virtual machines and the process of executing your recipes against those virtual machines. And second, by allowing you to capture and automate the work that was previously performed in ad hoc verification.

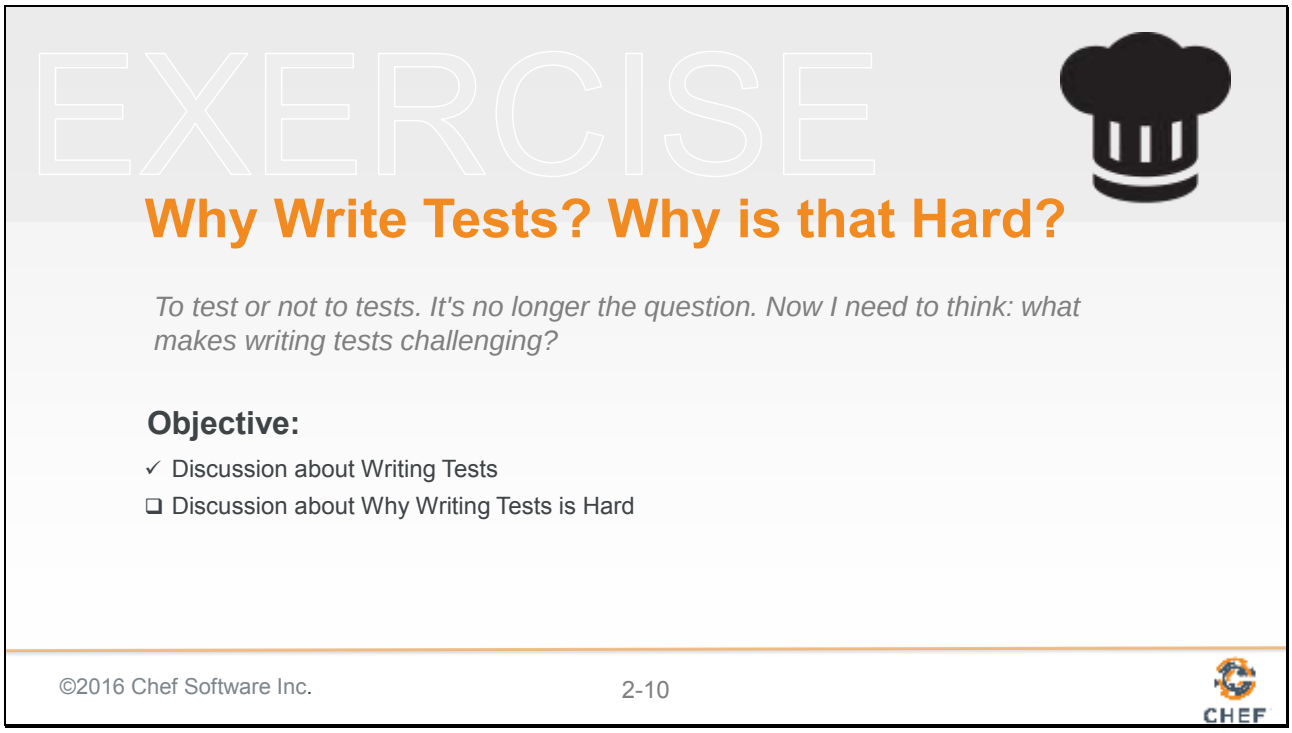
Slide 9



I shared with you my opinion on why I think it is important to write tests. Now I would like to understand what reasons you see for writing tests. I would also like to know your reasons for not writing tests.

Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 10



You may or may not be convinced that there is value in writing and executing tests. If the opinions we all have expressed has not convinced you I encourage you to continue to find more discussions where you can hear more opinions and share yours with others. It is important to have these discussions within your teams and your organization.

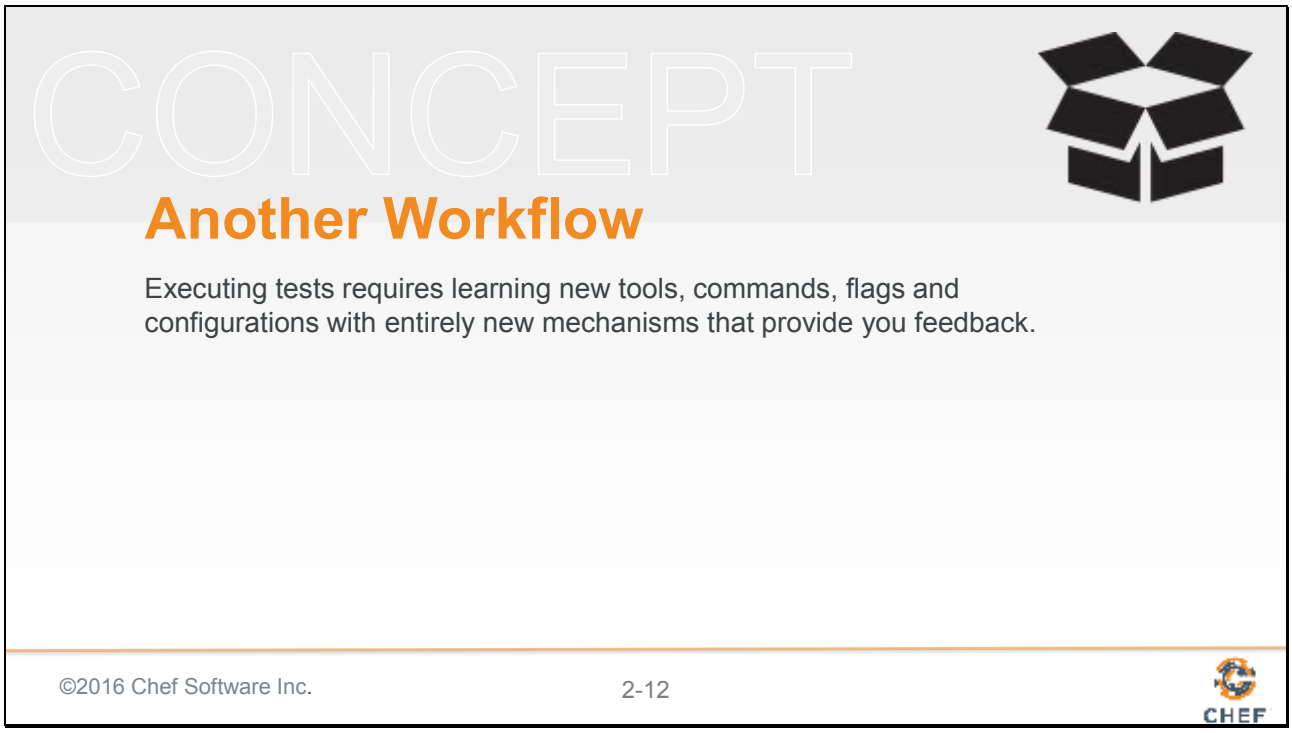
I want to now focus the discussion on the reasons why writing tests are hard. Similar to the previous discussion I want to provide my opinion to start the discussion. I want you to also contribute your opinions and experiences as they are equally valuable.

Slide 11

The language you use to define your tests in is not the same as the language you use to compose your original intentions. To test your code you need to write more code. However, this new code that you write is different as you are expressing your desired expectations of the system across a number of scenarios. This requires you to learn one or more new languages which have completely new systems and structures.

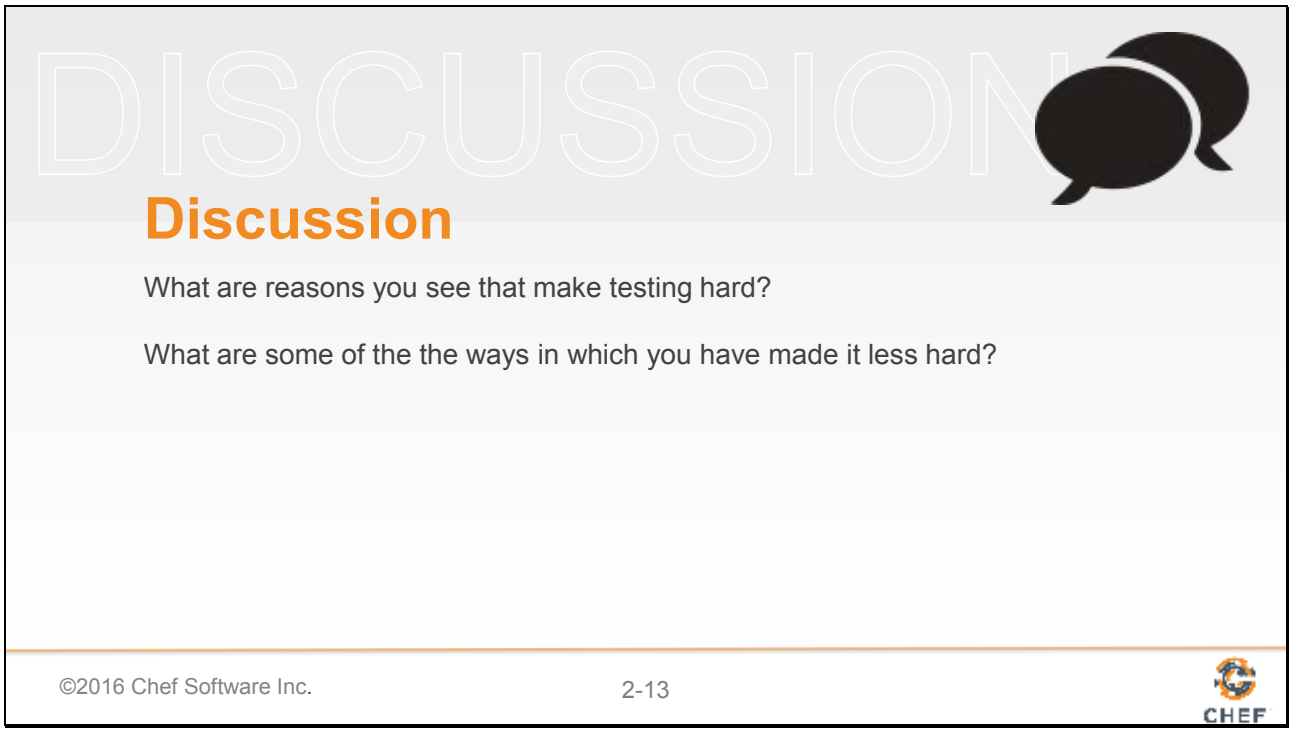
Testing asks you to solve a different problem in a different order when compared to process of writing software. You have to overcome particular challenges created by an implementation and express the desired expectations of that implementation before it is even built.

Slide 12



Testing also asks you to change your behaviors through the new tools required to execute the tests. These tools represent a huge domain of knowledge expressed in all the commands, flags, and configuration that must be understood to be used correctly and then effectively as the complexity of your testing tools grow. The largest, and most immediate impact is on your development workflow which has to adopt new steps that feel unsure and even more unreliable as you receive a barrage of feedback in unfamiliar formats.

Slide 13

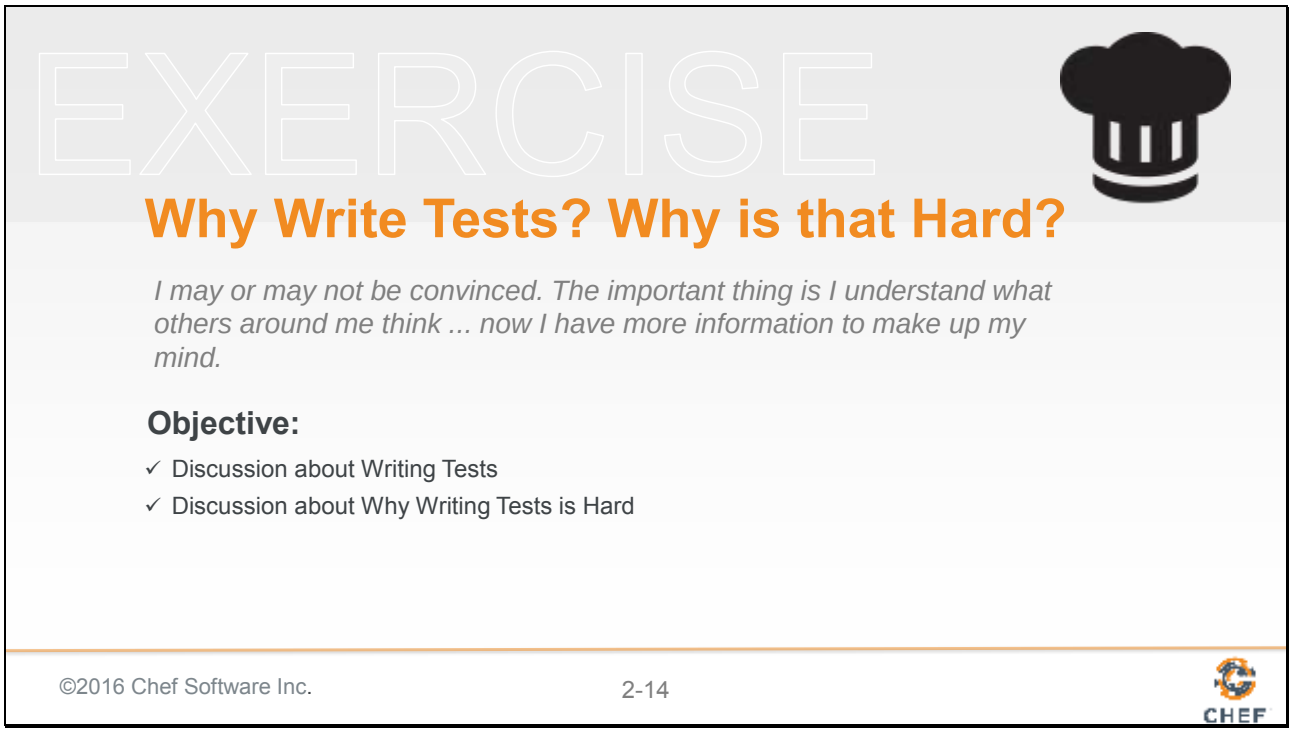


I shared with you my opinion on why I think it is hard to write tests. Now I would like to understand what reasons you see that make testing hard.

After we have expressed a set of reasons we should leave time within the discussion to discuss ways in which you have made it less hard.

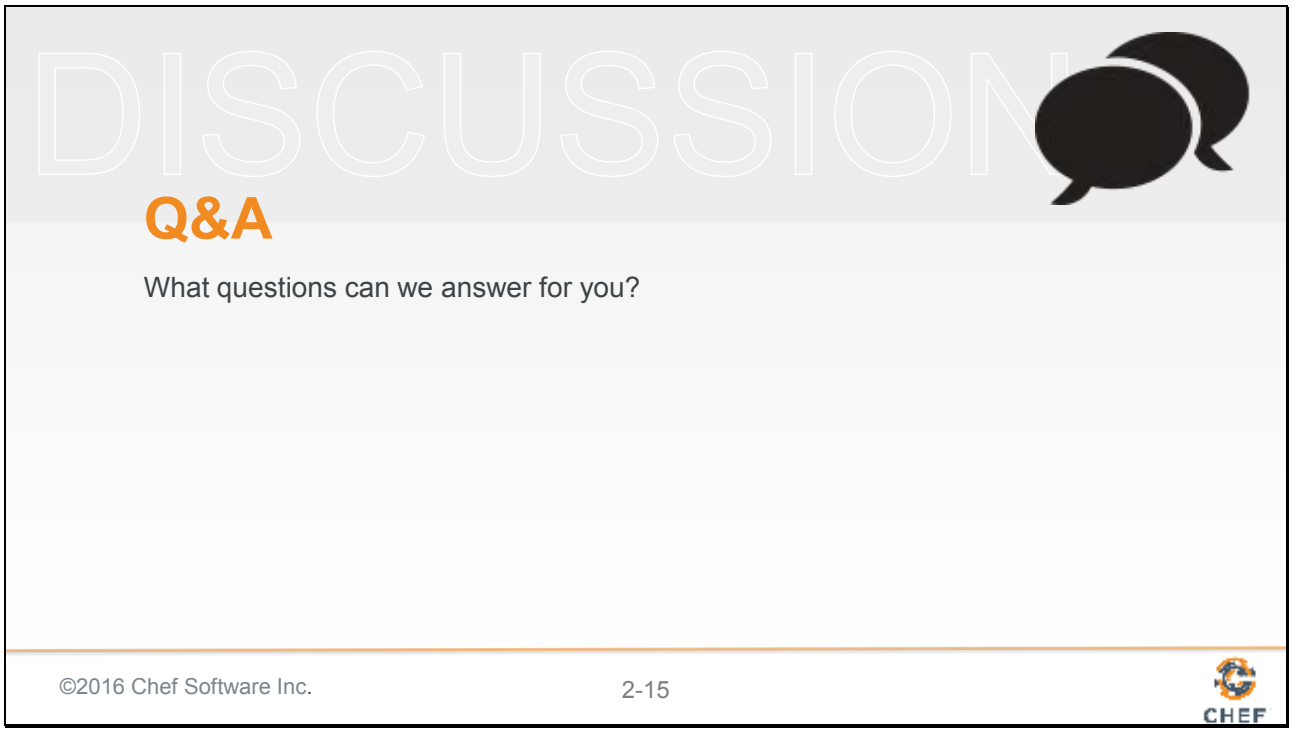
Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 14



With our two discussions complete let's pause now for any questions that were not covered or even came out of the discussions.

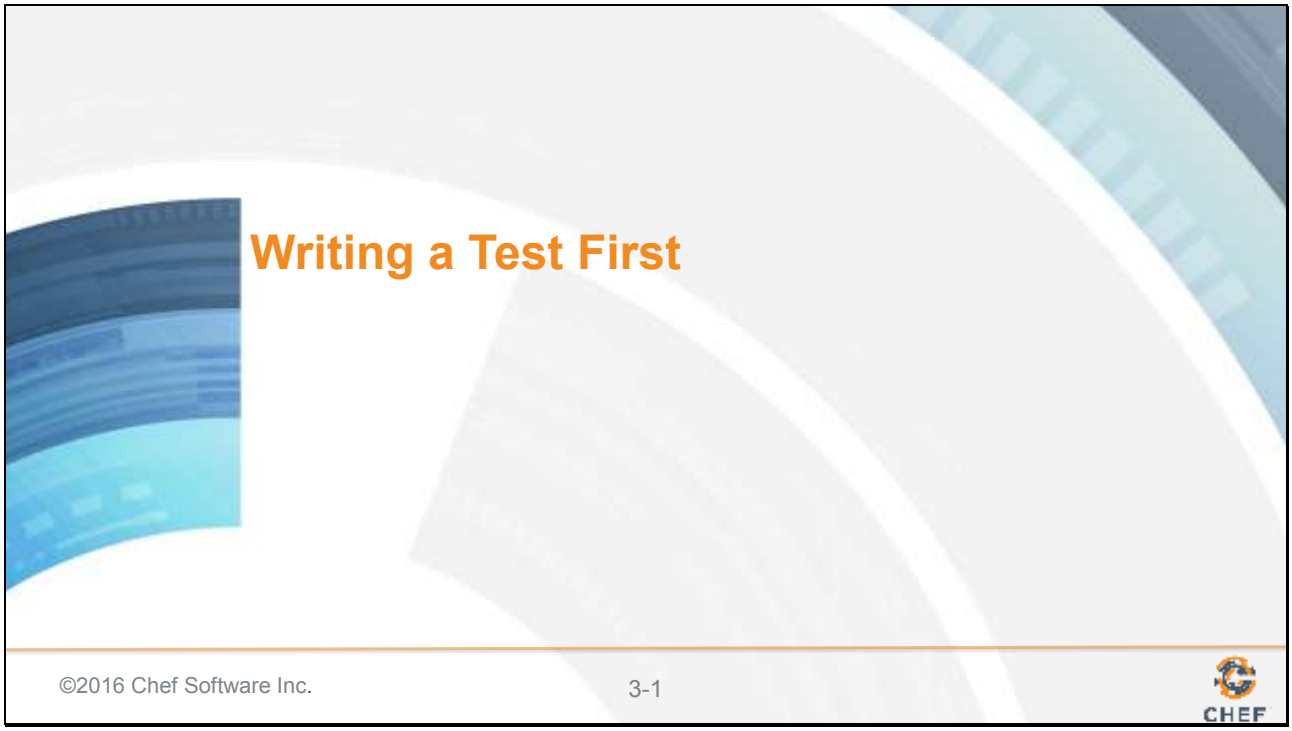
Slide 15



Before we complete this section and start learning some of these new tools and languages let us pause for questions.

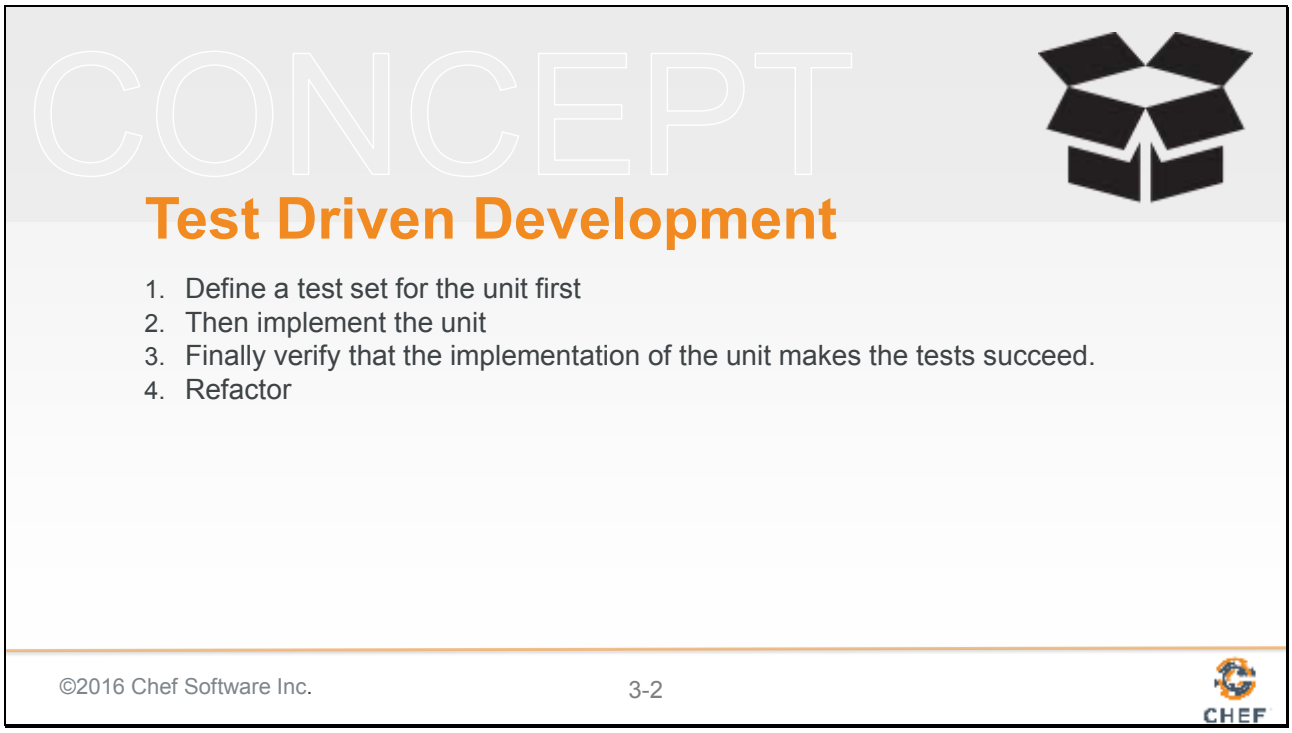
Slide 16

3: Writing a Test First



Writing tests are often difficult. Writing tests before you have written the code that you want to test can often feel like a leap of faith. An act that requires a level of clairvoyance reserved for magicians or con-artists. Some have likened it towards starting a story by first writing the conclusion.

Slide 2

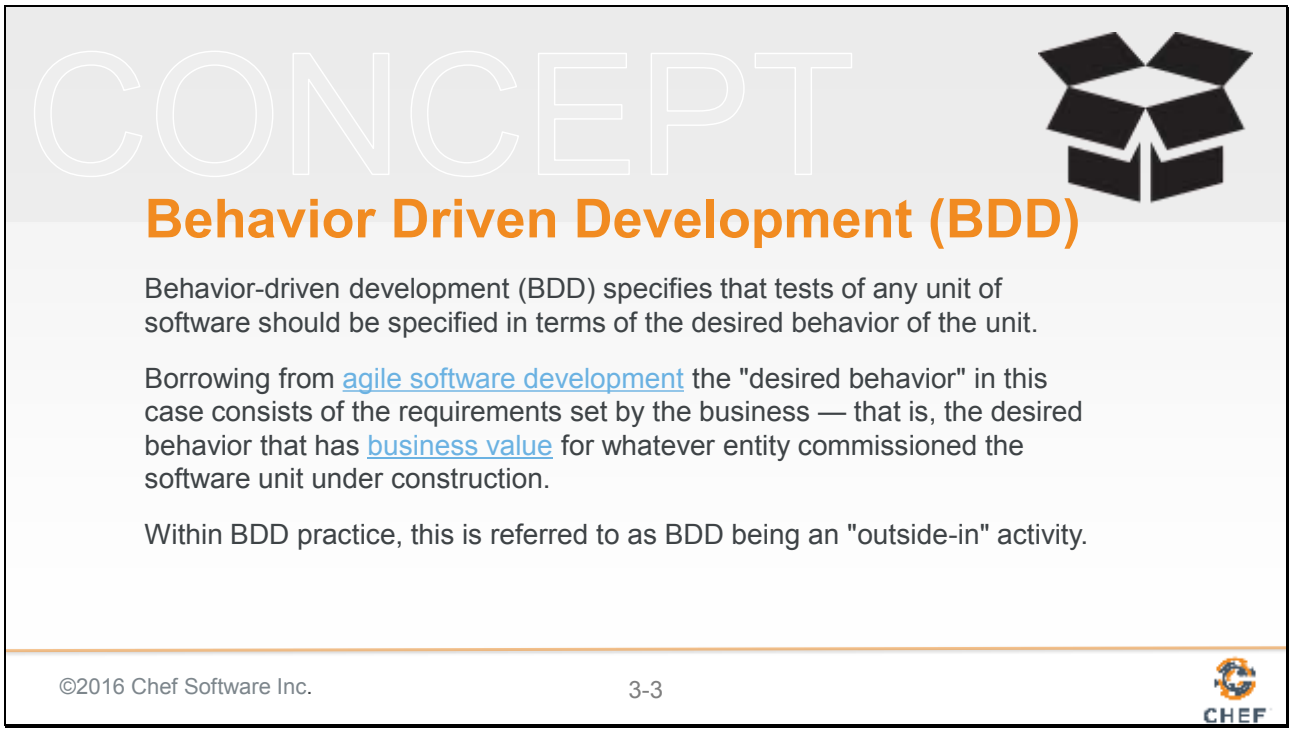


Test Driven Development (TDD) is a workflow that asks you to perform that act continually and repeatedly as you satisfy the requirements of the work you have chosen to perform.

TDD generically focuses on the unit of software any level. It is the process of writing the test first, implementing the unit, and then verifying the implementation with the test that was written.

A 'unit' of software is purposefully vague. This 'unit' is definable by the individuals developing the software. So the size of a 'unit of software' likely has different meanings to different individuals based on our backgrounds and experiences.

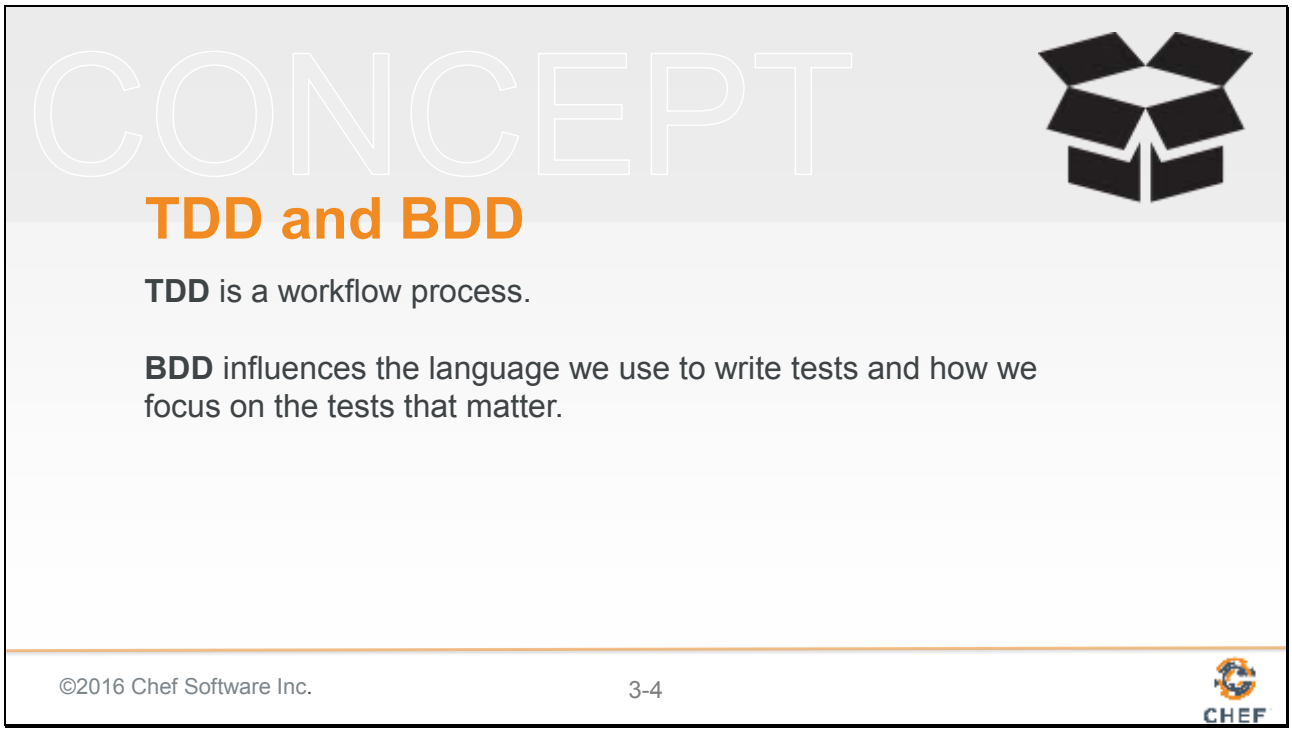
Slide 3



How you choose to express the requirements of that unit is the crux of Behavior Driven Development (BDD). Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit. Expressing this desired behavior is often expressed in scenarios that are written in a Domain Specific Language (DSL).

The cookbooks and recipes that you have written so far share quite a few similarities with BDD. In Chef, you express the desired state of the system through a DSL, resources, you define in recipes.

Slide 4



TDD is a workflow process: Add a test; Run the test expecting failure; Add code; Run the test expecting success. Refactor.

BDD influences the language we use to write the tests and how we focus on tests that matter. The activities within this module focus on the process of taking requirements, expressing them as expectations, choosing one implementation to meet these expectations, and then verifying we have met these expectations.

Slide 5

Objectives

After completing this module, you should be able to:

- Use chef to generate a cookbook
- Write an integration test
- Use Test Kitchen to create, converge, and verify a recipe
- Develop a cookbook with a test-driven approach

In this module you will learn how to use chef to generate a cookbook, write an integration test first, use Test Kitchen to execute that test, and then implement a solution to make that test pass.

Slide 6

Building a Web Server

1. Install the httpd package
2. Write out a test page
3. Start and enable the httpd service

To explore the concepts of Test Driven Development through Behavior Driven Design we are going to focus on creating a cookbook that starts with the goal that installs, configures, and starts a web server that hosts the your company's future home page.

This cookbook will start very straight-forward and over the course of these modules we will introduce new requirements that will increase its complexity.

The goal again is to focus on the TDD workflow and understanding how to apply BDD when defining these tests. We are not concerned about focusing on best practices for managing web servers or modeling a more initially complex cookbook.

Slide 7

Defining Scenarios

Given SOME CONDITIONS

When an EVENT OCCURS

Then I should EXPECT THIS RESULT

When requirements come to us it is rare that the product owners and customers ask us to deliver a particular technology or a software. In our case, I have asked you to setup a web page for your company. I did not specifically state a particular technology but to help limit the scope I have chosen that we are going to build this initial website with Apache.

Behavior driven design asks us to look at the work that we perform from the perspective of our users. Our first job is to develop the scenario that validates the work that we are about to accomplish.

These scenarios that we write are often written in the following format.

This very generically defines any scenario. What we need to do is apply this scenario format to our requirements.

Slide 8

The Why Stack?

You should discuss...the feature and pop the why stack max 5 times (ask why recursively) until you end up with one of the following business values:

- Protect revenue
- Increase revenue
- Manage cost

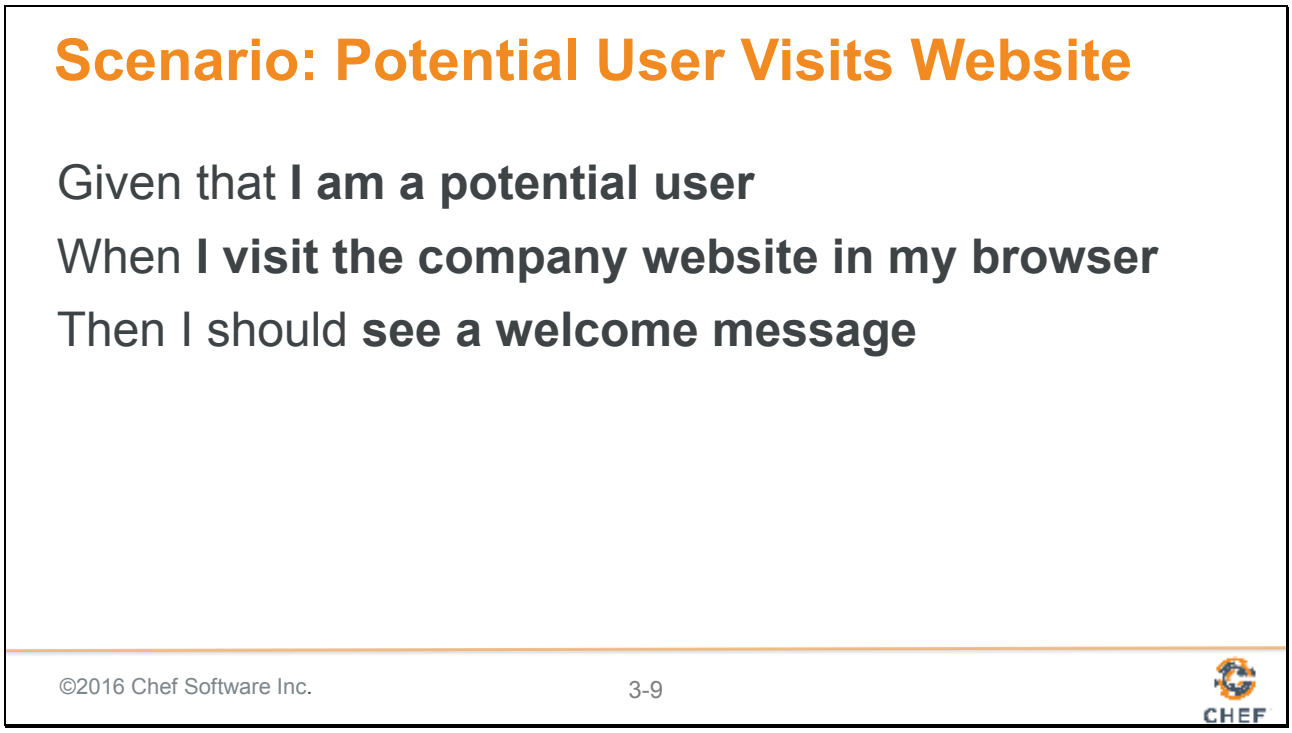
If you're about to implement a feature that doesn't support one of those values, chances are you're about to implement a non-valuable feature. Consider tossing it altogether or pushing it down in your backlog.

- Aslak Hellesøy, creator of Cucumber

If our goal is to setup a new webpage we need to start to ask ourselves the question: Why. Why do we need to setup a website? Asking this question will help us identify for who the website is for and what purpose does it serve for the actor in this scenario.

Often times the why will raise more questions which you continue to ask why. You should do that. Asking why enough times will lead you to the true reason why you are taking action. The interesting thing is that knowing the true reason why will help reinforce your course of action or maybe change it entirely.

Slide 9



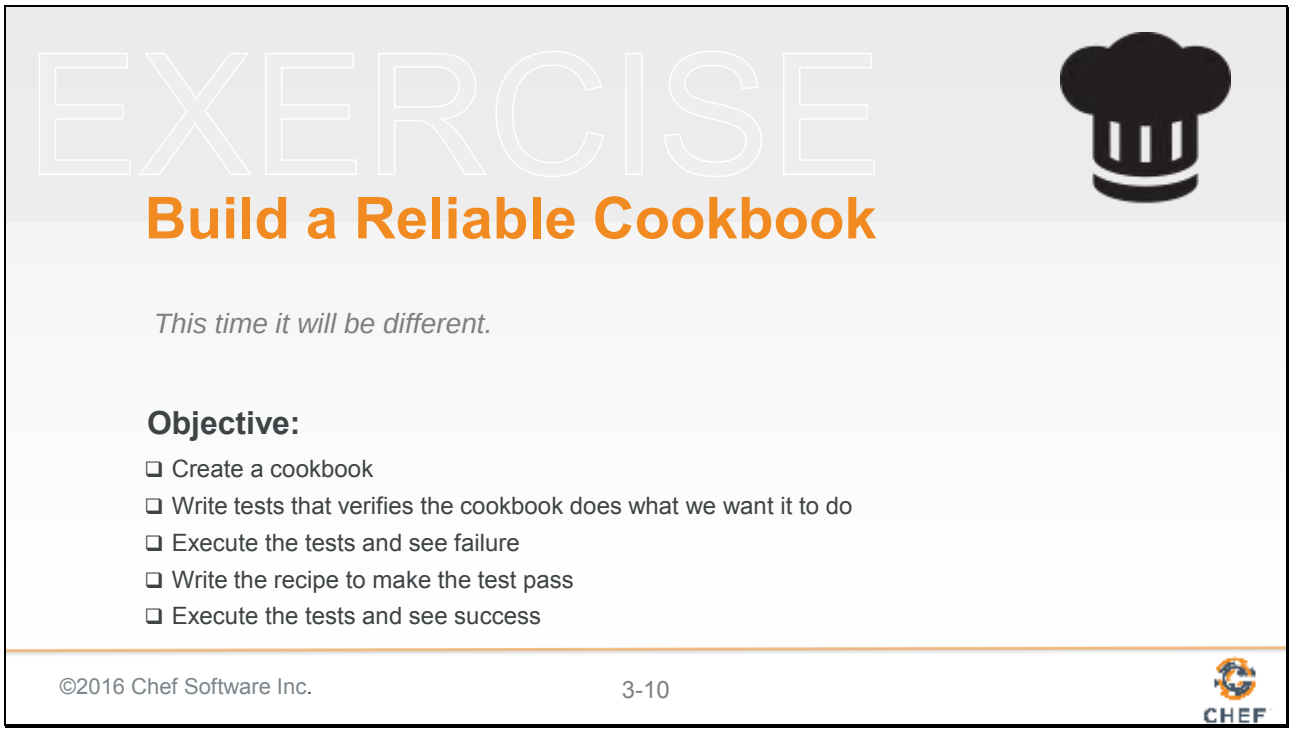
The typical reason for setting up a website is to allow customers, users, potential users to learn more about the company. The needs of the website may change in the future but the first minimum viable product (MVP) is to simply give our users the ability to find out more information.

Our goal now is to define a scenario with this understanding.

This first scenario is enough information to help us build this cookbook with a TDD approach. This practice of defining a scenario is a tactic that I employ to help focus me on the most valuable work that needs to be done.

Important things to notice in the following scenario is the distinct lack of technology or implementation. The scenario is not concerned about the services that are running or files that might be found on the file system.

Slide 10

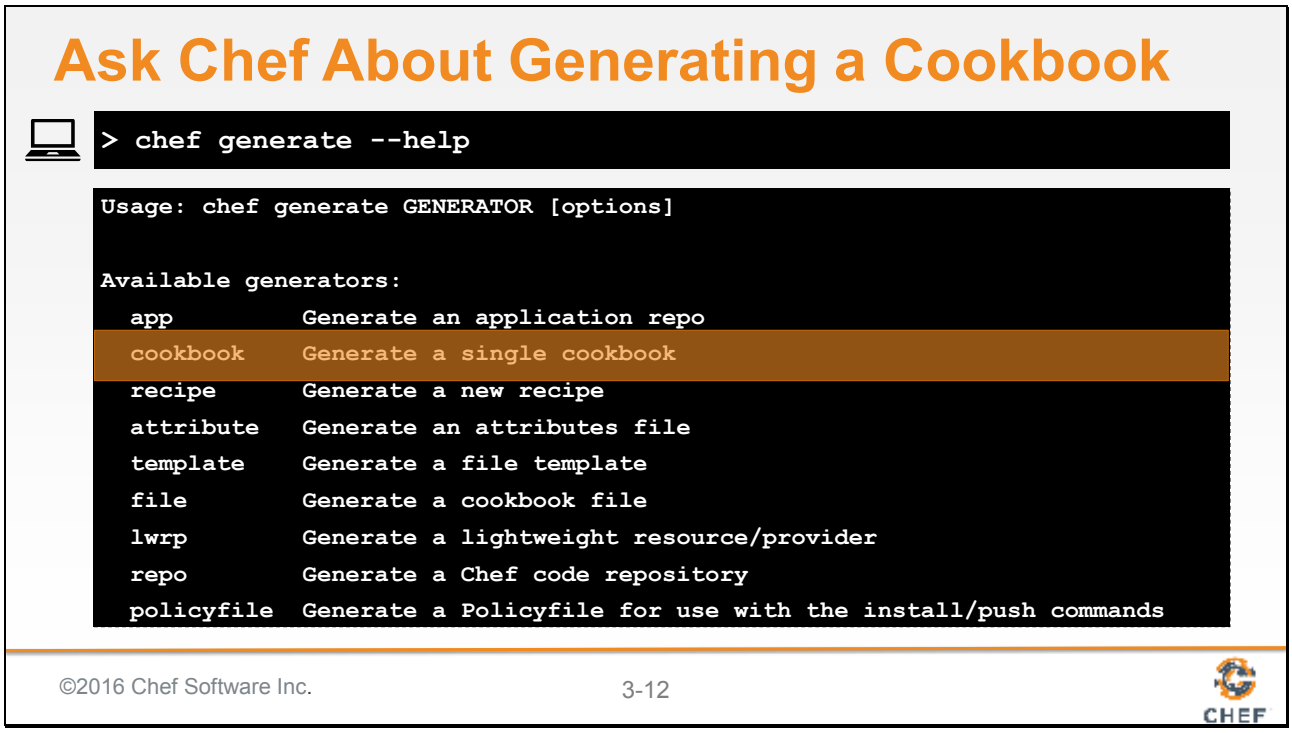


With the scenario defined it is now time for us to develop the cookbook. We are going to move through the following steps together to accomplish this task.

Slide 11

Let's start the journey on your workstation. From the home directory we are going to creating this cookbook.

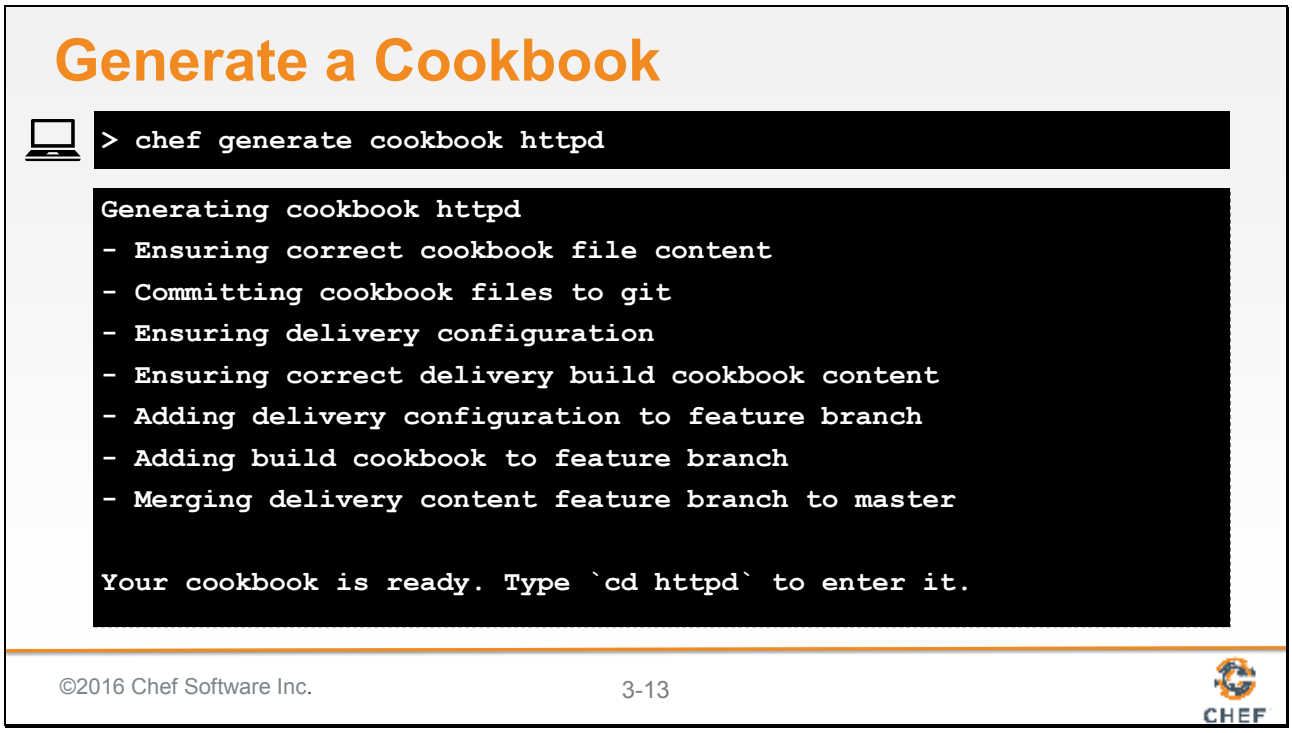
Slide 12



There are a number of tools installed with the Chef Development Kit (Chef DK). One of those tools included in the Chef DK is a tool called 'chef'. The generators provided with the tool will allow us to quickly generate the a cookbook. You can see help about the command with the '--help' flag.

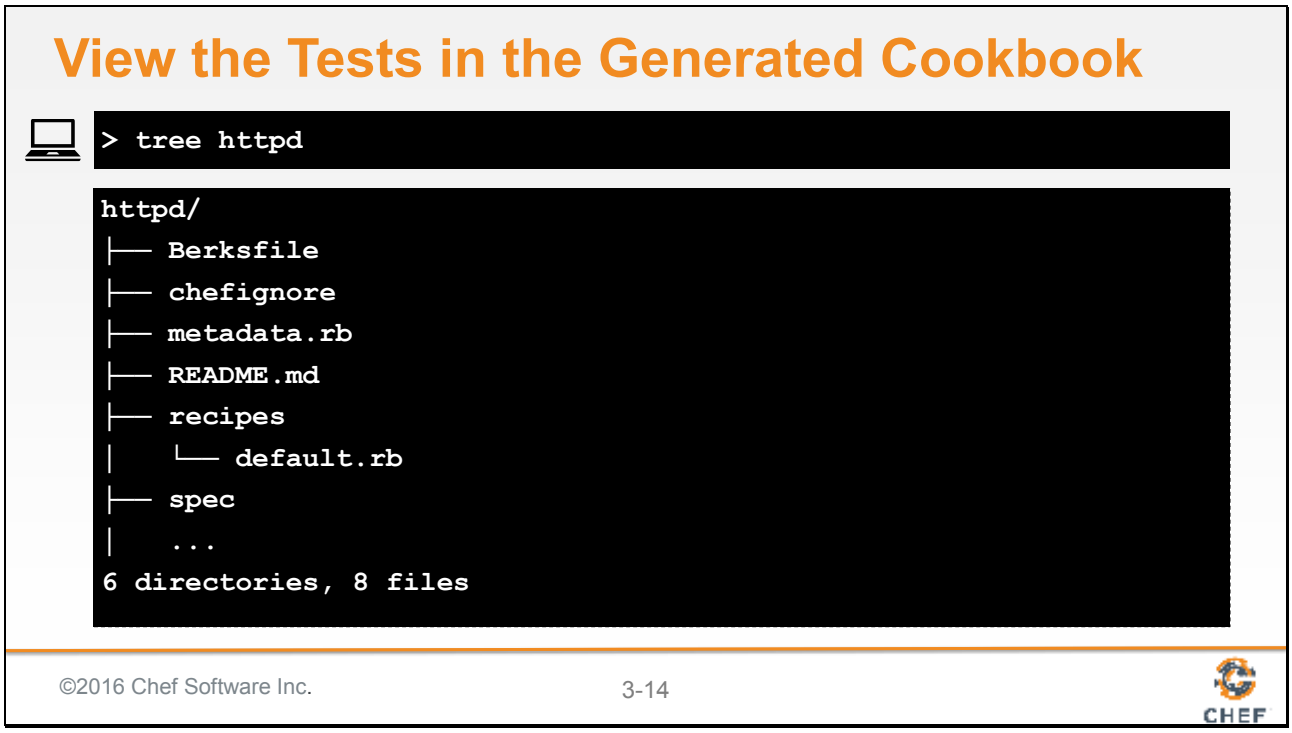
The cookbook generator has only one required parameter and that is the name of the cookbook.

Slide 13



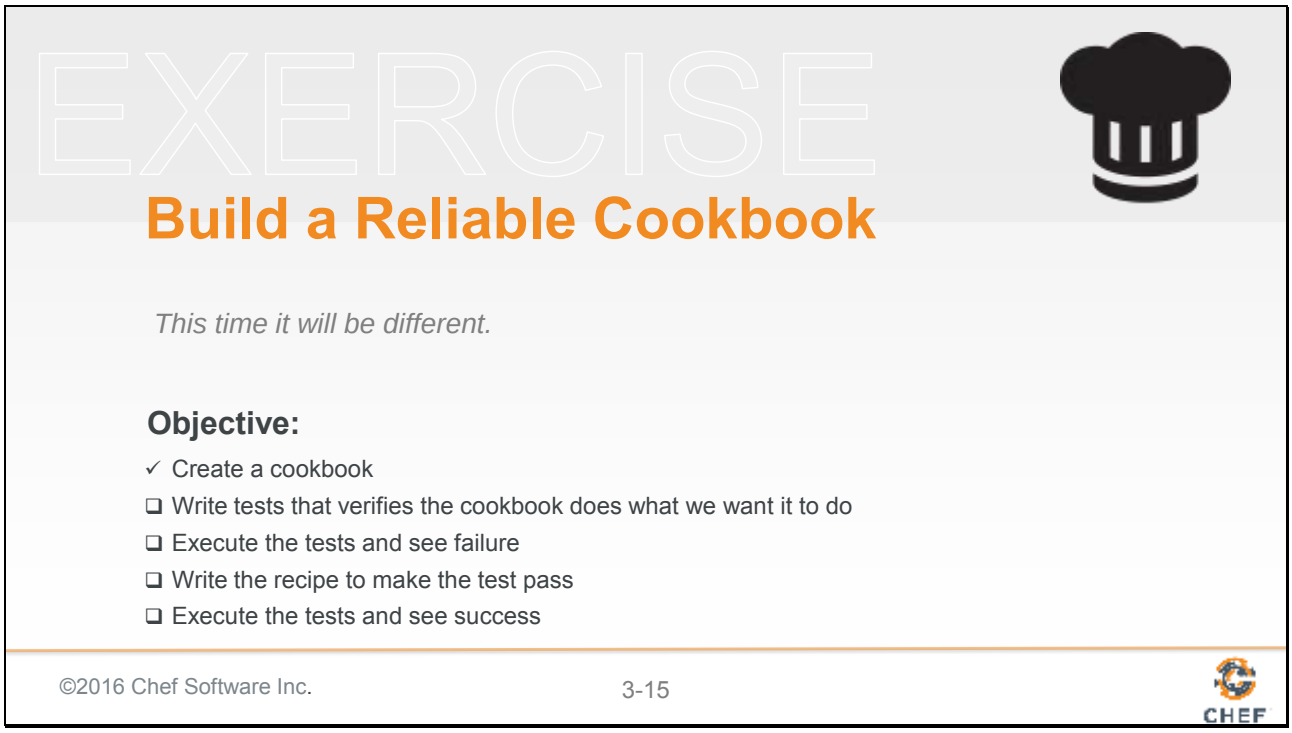
Let's generate cookbook named 'httpd'. The name of the cookbook here resembles the name of a public cookbook in the Supermarket that accomplishes a very similar task. That is the reason why I have asked you to chose the same name.

Sharing the same name as a cookbook within the Supermarket can be problematic. While we may never share this cookbook other individuals within our organization could believe it to be a copy of that cookbook. When it comes to naming cookbooks it may be wise to first search the Supermarket and ensure you are not using a similar name.



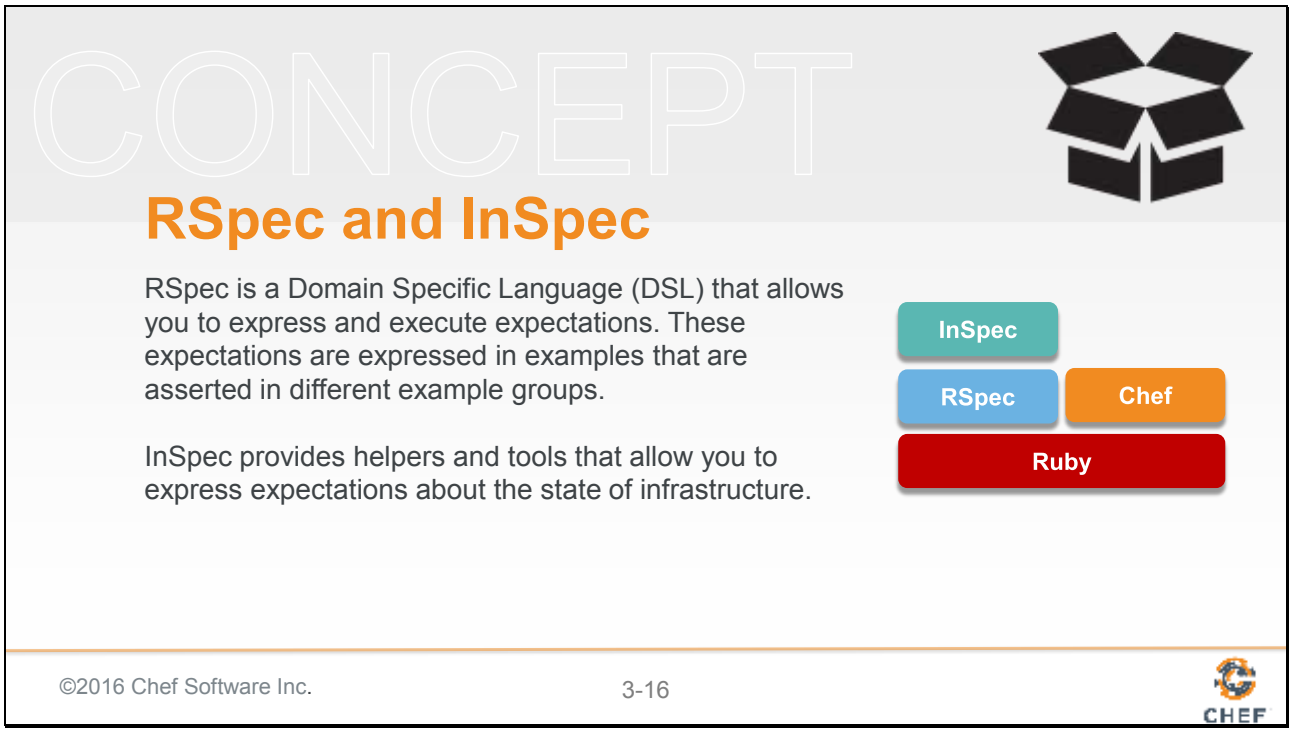
We can examine the contents of the cookbook that chef generated for us. Here you see that the tool created for us a complete test directory structure.

Slide 15



With the cookbook created it is now time to write that first test that verifies the cookbook does what we want it to do.

Slide 16



RSpec is a Behavior Driven Development (BDD) framework that uses a natural language domain-specific language (DSL) to quickly describe scenarios in which systems are being tested. RSpec allows you to setup a scenario, execute the scenario, and then define expectations on the results. These expectations are expressed in examples that are asserted in different example groups.

RSpec by itself grants us the framework, language, and tools. InSpec provides the knowledge about expressing expectations about the state of infrastructure.

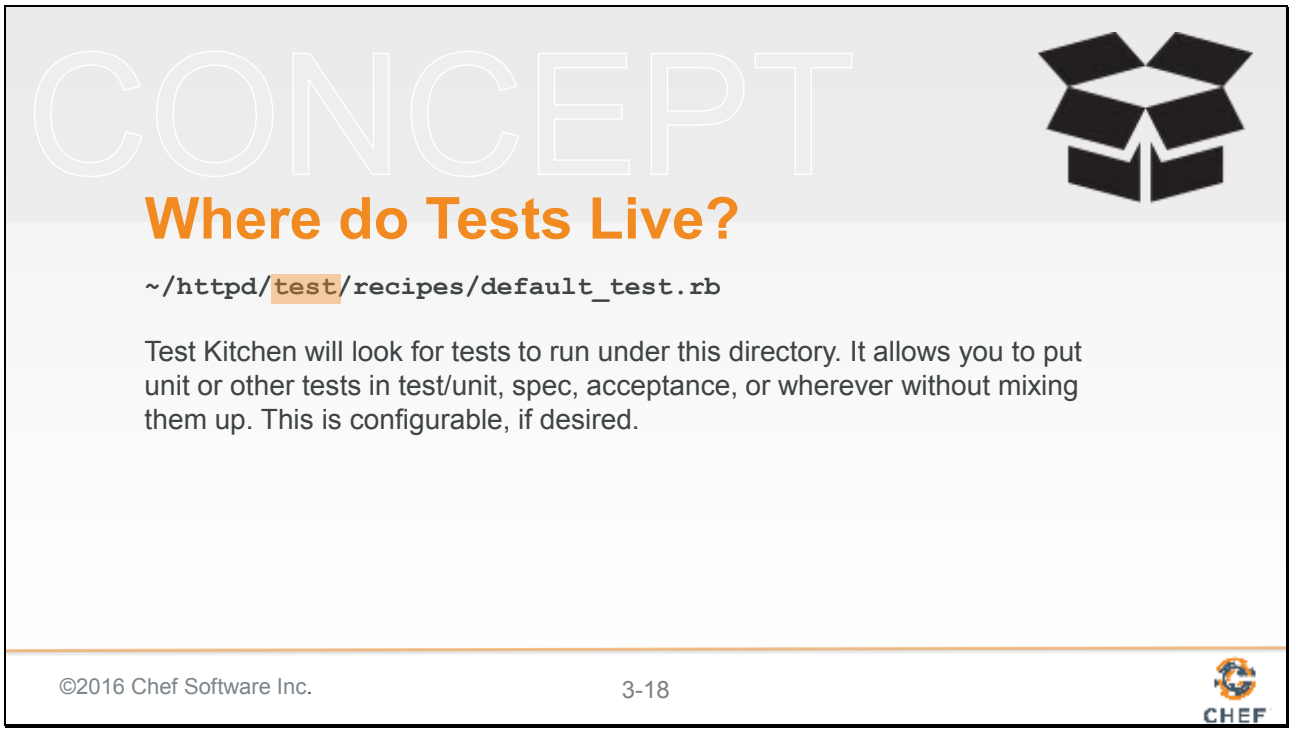
Auto-generated Spec File in Cookbook

~/httpd/test/recipes/default_test.rb

```
unless os.windows?
  describe user('root') do
    it { should exist }
    skip 'This is an example test, replace with your own test.'
  end
end

describe port(80) do
  it { should_not be_listening }
  skip 'This is an example test, replace with your own test.'
end
```


The generator created an example specification (or spec) file. Before we talk about the RSpec/InSpec language lets explain the long file path and its importance.

Slide 18

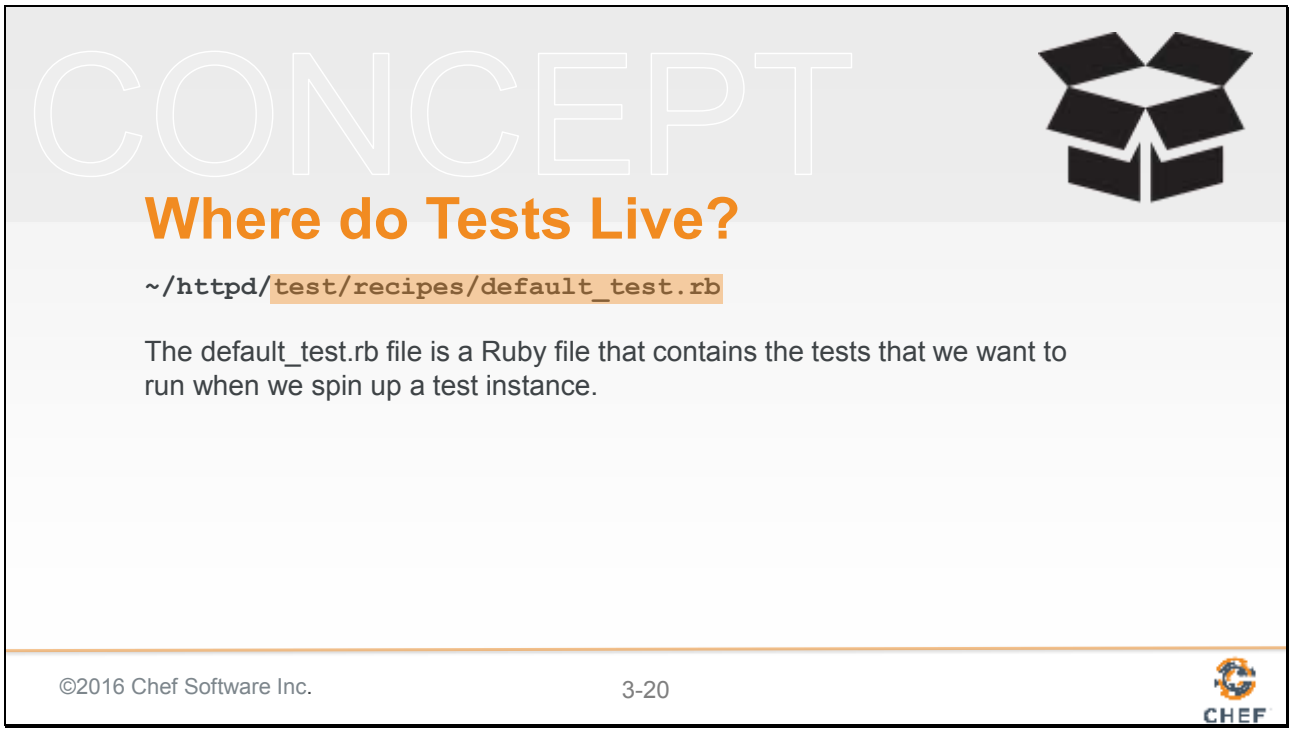


Let's take a moment to describe the reason behind this directory path. Within our cookbook we define a test directory and within that test directory we define another directory named 'recipes'. This is the basic file path that Test Kitchen expects to find the specifications defined in InSpec.

Slide 19

The next part the path, 'recipes', corresponds to the path specified in the .kitchen.yml file.

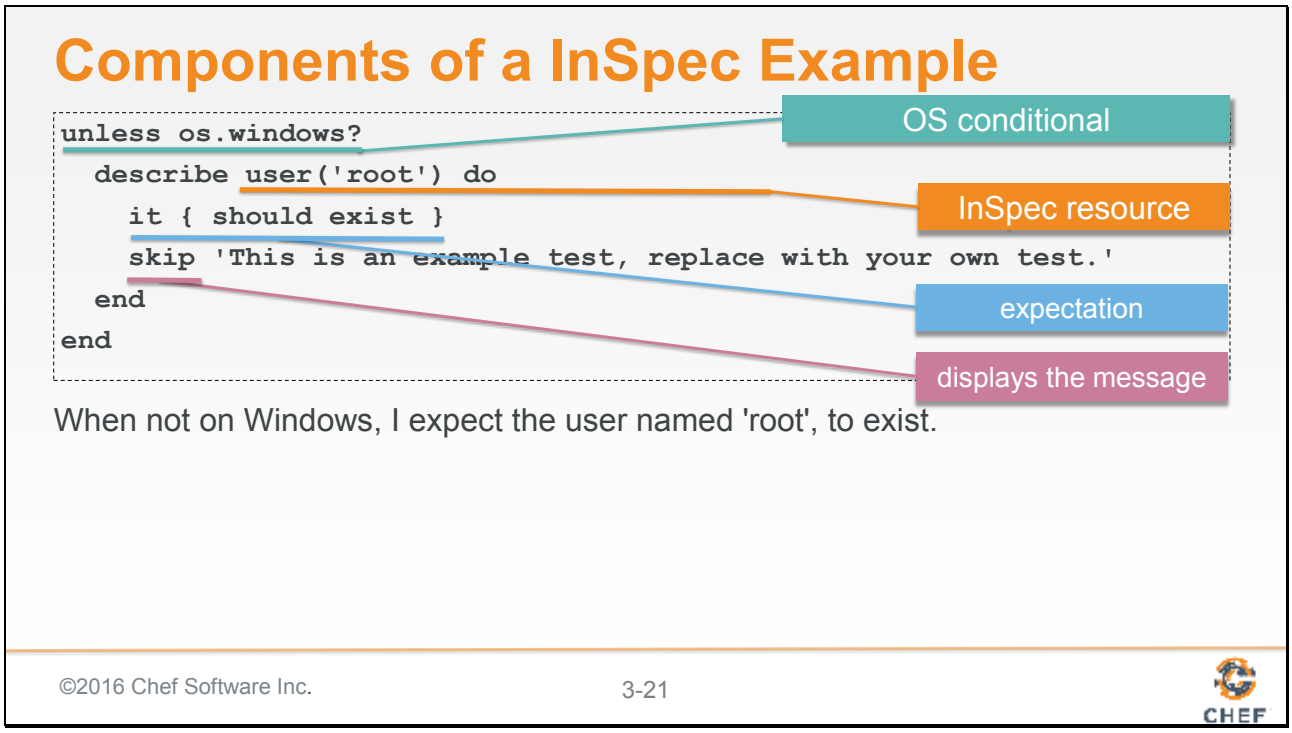
Slide 20



The ruby file, default_test.rb, contains the tests that we have defined. A test file is a Ruby file that contains domain specific language constructs that we use to express our desired state of the system.

Let's open this default_test.rb file and review the contents of it.

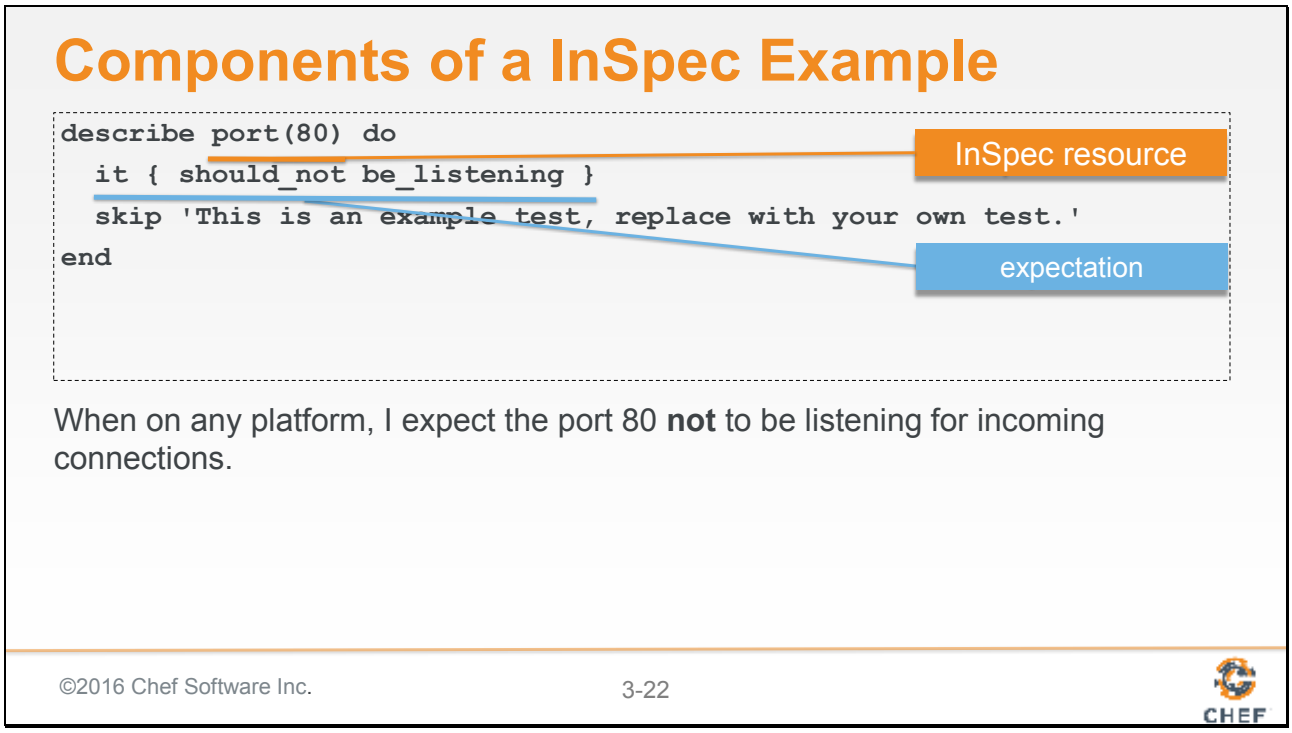
Slide 21



The outermost statement is a conditional that states that when we want to evaluate the contents in between when we are not on Windows (e.g. CentOS, Ubuntu, Debian).

The inner describe has two parameters: The first is the the user resource named 'root' on the test instance. The second is the block which contains the expectations that we want to assert for the given resource.

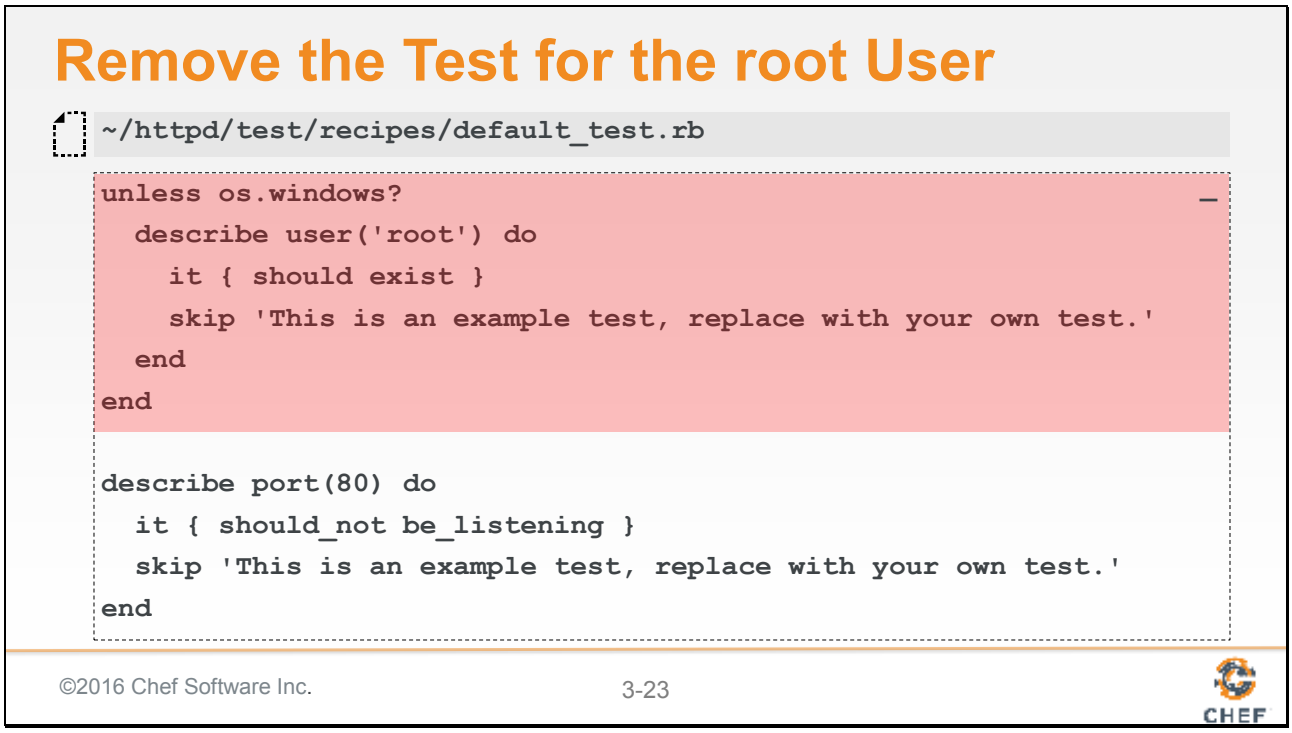
Within the block we can define any number of expectations about the particular resource in the description. In this instance we are saying that we expect the user, named 'root', to exist on the instance. After the expectation that has been defined is a skip. This skip is a reminder that the examples have been defined in this test file were automatically generated and should be updated or removed.



The second example within the test file describes the port 80 on any operating system and states the expectation that it does not expect port 80 to be listening.

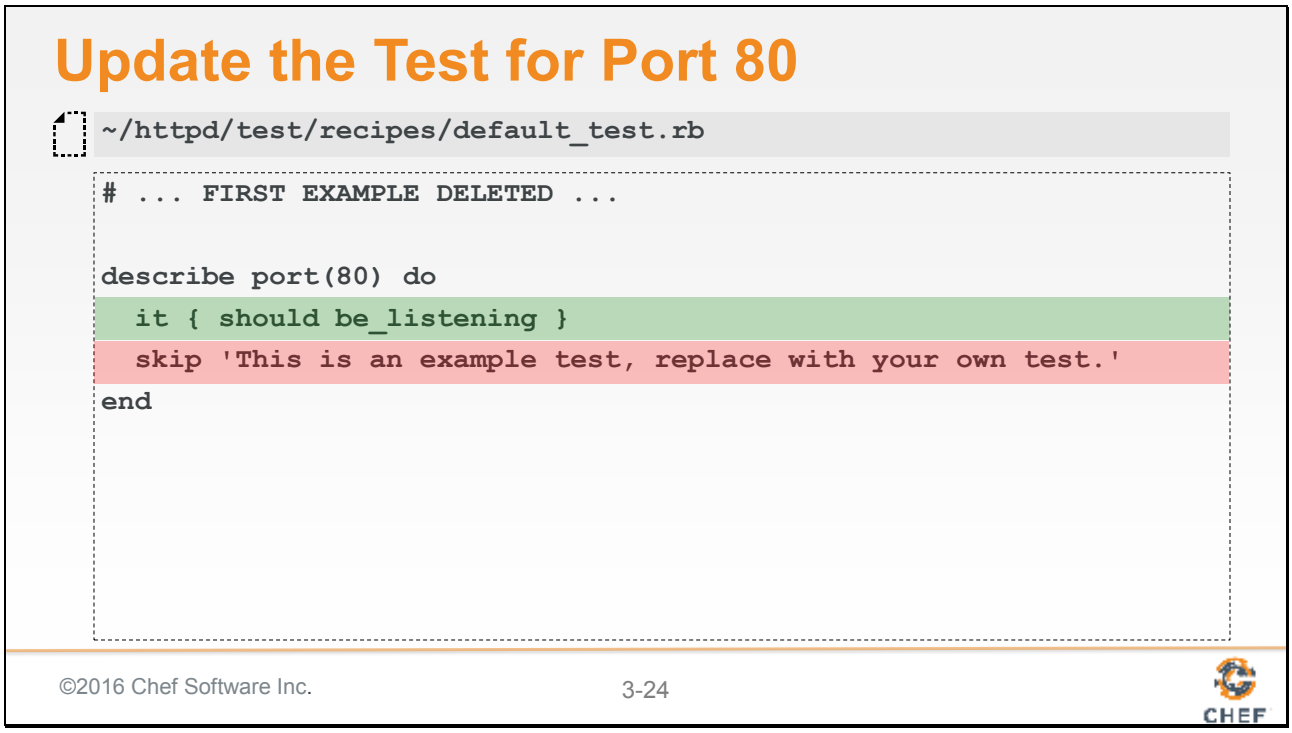
By default all operating systems will be examined. So this example would be evaluated and executed against every operating system.

Slide 23



Within the test file found at the following path you will find that it is already populated with that initial code. Remove the first test that asserts that the user named root exists on the system. While it is likely true we are not interested in verifying that with this cookbook.

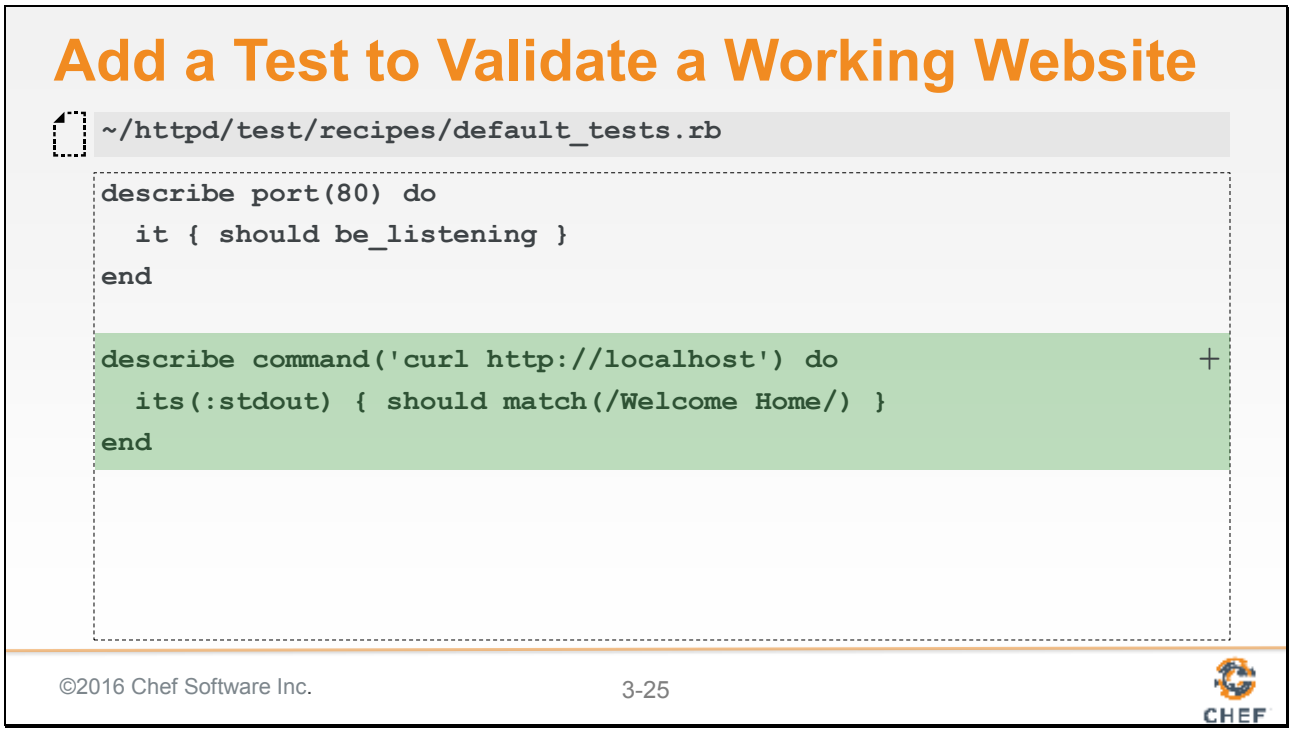
Slide 24



The other expectation expressed within this file is useful but it is wrong. When we setup a web server we are going to want to have incoming connections on port 80.

So update the following example to state that port 80 should be listening. Also remove the line with the skip as we have now successfully updated the test and I would consider it one that is ours and correct for the code we will eventually write.

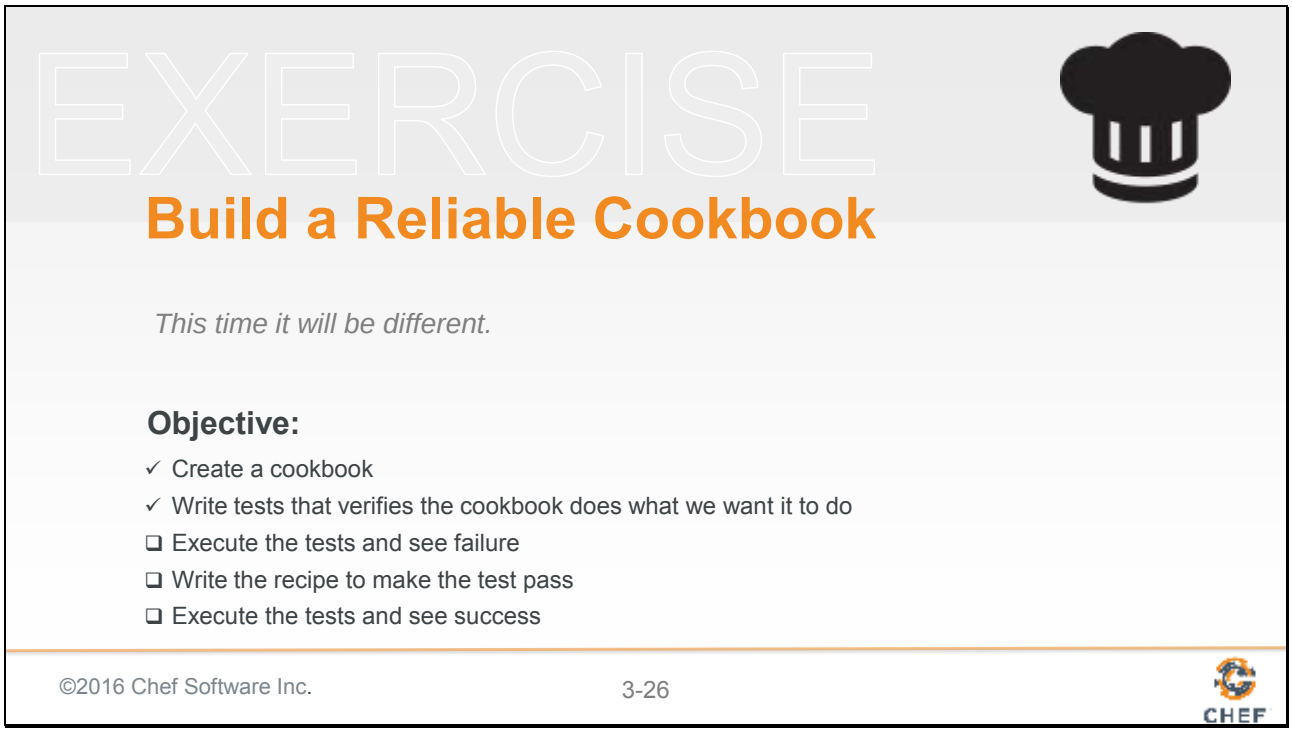
Slide 25



Ensuring that we are listening on port 80 for incoming connections does not verify that we are in fact returning the correct home page with the welcoming message we plan to write. To do that we will need to write a new expectation.

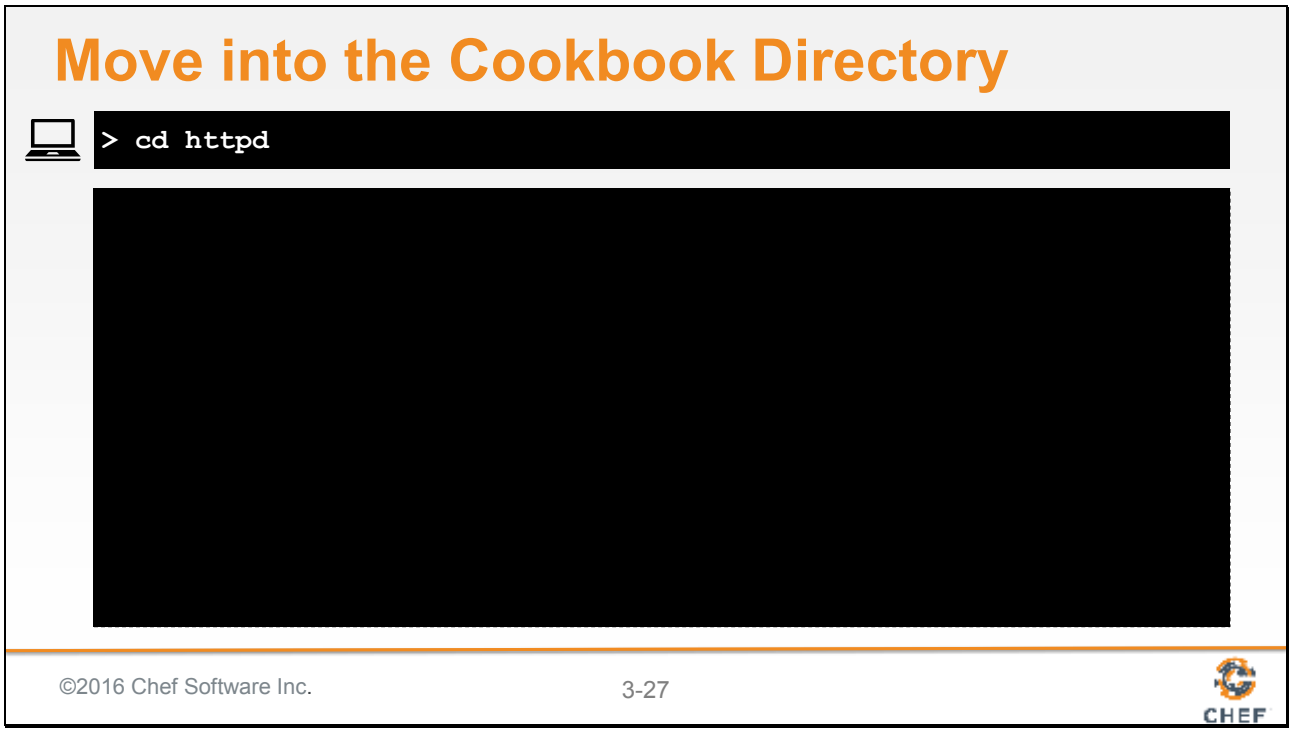
InSpec provides a helper method that allows you to specify a command. That command returns the results from the command through standard out. We are asking the command's standard out if anywhere in the results match the value 'Welcome Home'.

Slide 26



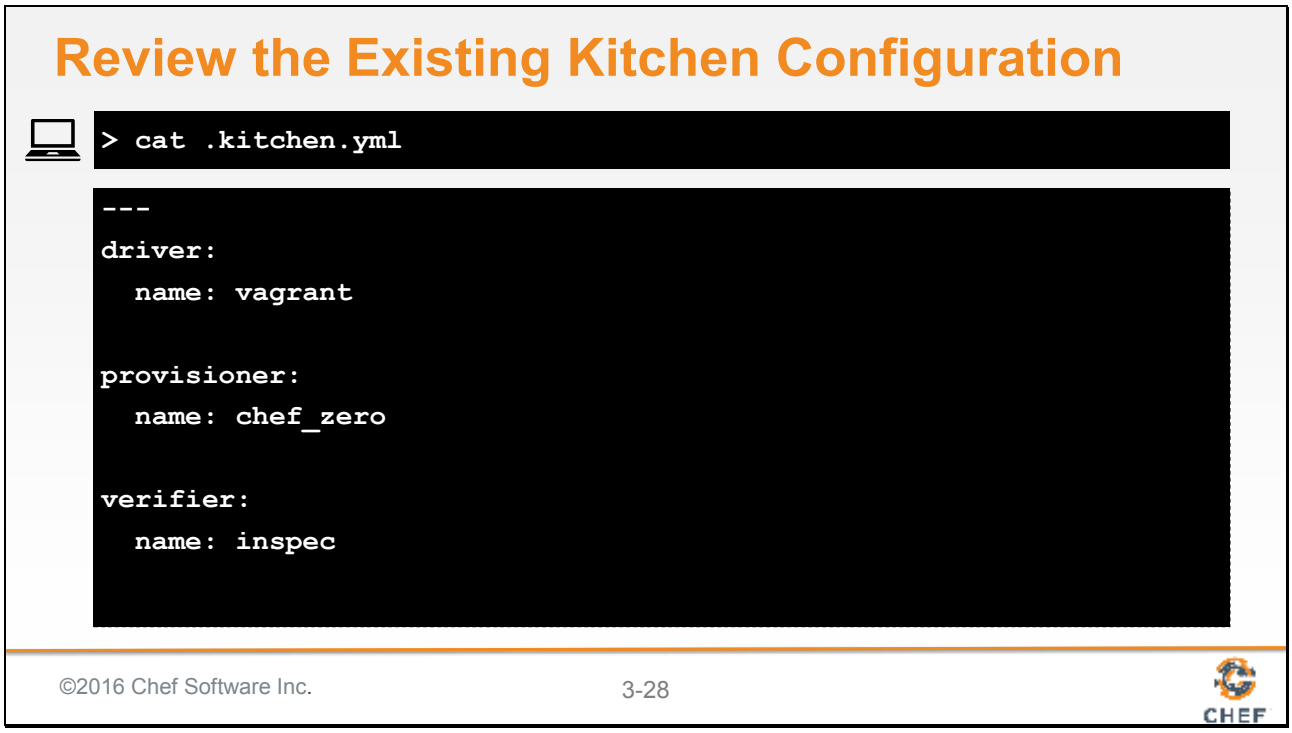
With the test defined it is now time to execute the tests and see the failure.

Slide 27



To execute our tests using the tool Test Kitchen we need to be within the directory of the cookbook.

Slide 28



Before we employ Test Kitchen to execute the tests we need make changes to the existing Test Kitchen configuration file. The cookbook was automatically generated with a '.kitchen.yml'.

Slide 29

The Kitchen Driver

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

verifier:
  name: inspec

platforms:
  - name: ubuntu-16.04
  - name: centos-7.2
```

The driver is responsible for creating a machine that we'll use to test our cookbook.

Example Drivers:

- docker
- vagrant

The first key is driver, which has a single key-value pair that specifies the name of the driver Kitchen will use when executed.

The driver is responsible for creating the instance that we will use to test our cookbook. There are lots of different drivers available--two very popular ones are the docker and vagrant driver.

Instructor Note: Testing on this remote workstation requires that we use Docker because Vagrant does not work within a virtual environment. Vagrant is the standard choice when working on your local workstation.

The Kitchen Provisioner

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

verifier:
  name: inspec

platforms:
  - name: ubuntu-16.04
  - name: centos-7.2
```

This tells Test Kitchen how to run Chef, to apply the code in our cookbook to the machine under test.

The default and simplest approach is to use chef_zero.

The second key is provisioner, which also has a single key-value pair which is the name of the provisioner Kitchen will use when executed. This provisioner is responsible for how it applies code to the instance that the driver created. Here the default value is chef_zero.

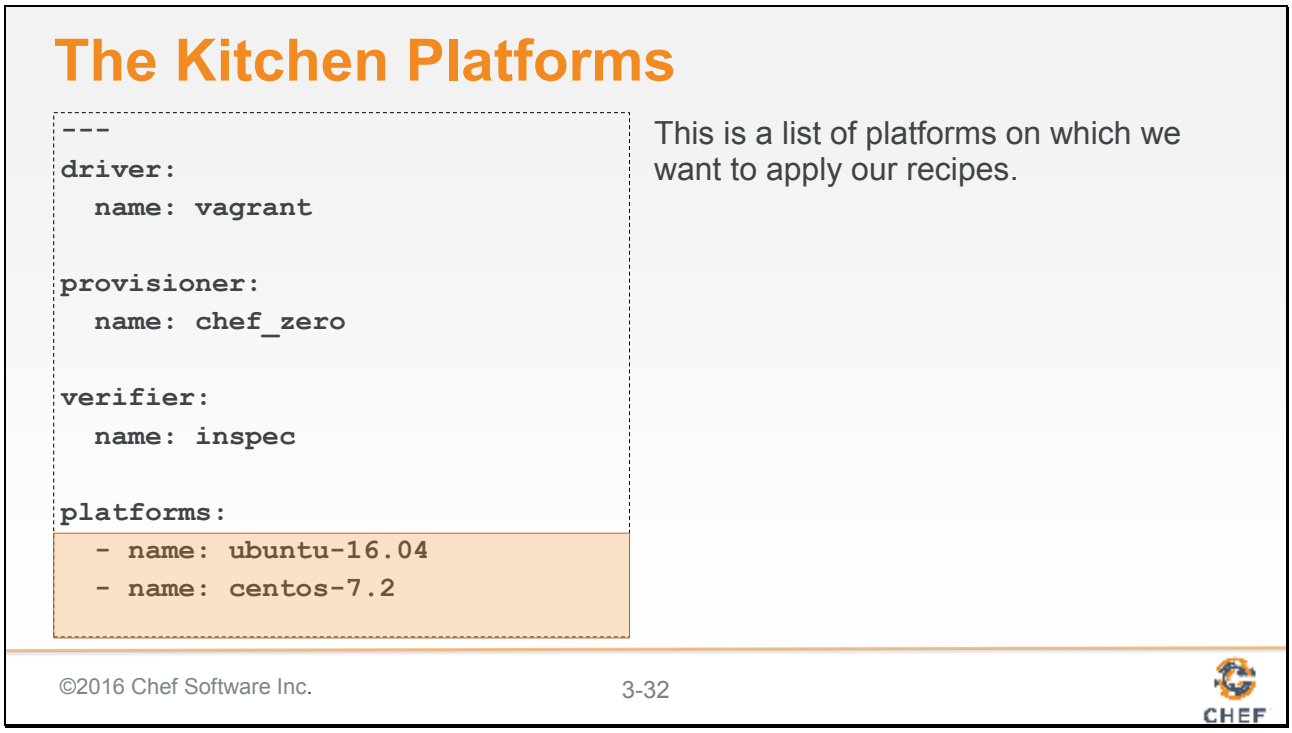
Slide 31

The Kitchen Verifier

```
---
driver:
  name: vagrant

provisioner:
  name: chef_zero

verifier:
  name: inspec

platforms:
  - name: ubuntu-16.04
  - name: centos-7.2
```

This is the framework that is used to verify the state of the system meets the expectations defined.

The third key is the verifier. This verifier by default is using InSpec. Test Kitchen has the ability to use several different verifiers. The default generated with the cookbook generator is InSpec.



The fourth key is platforms, which contains a list of all the platforms that Kitchen will test against when executed. This should be a list of all the platforms that you want your cookbook to support.

The Kitchen Suites

```
platforms:
  - name: ubuntu-16.04
  - name: centos-7.2

suites:
  - name: default
    run_list:
      - recipe[httpd::default]
    verifier:
      inspec_tests:
        - test/recipes
    attributes:
```

This section defines what we want to test. It includes the Chef run-list of recipes that we want to test.

We define a single suite named "default".

The fifth key is suites, which contains a list of all the test suites that Kitchen will test against when executed. Each suite usually defines a unique combination of run lists that exercise all the recipes within a cookbook.

In this example, this suite is named 'default'.

The Kitchen Suites' Run List

```
platforms:
  - name: ubuntu-16.04
  - name: centos-7.2

suites:
  - name: default
    run_list:
      - recipe[httpd::default]
    verifier:
      inspec_tests:
        - test/recipes
    attributes:
```

The suite named "default" defines a run_list.

Run the "httpd" cookbook's "default" recipe file.

CHEF

This default suite will execute the run list containing: The httpd cookbook's default recipe.

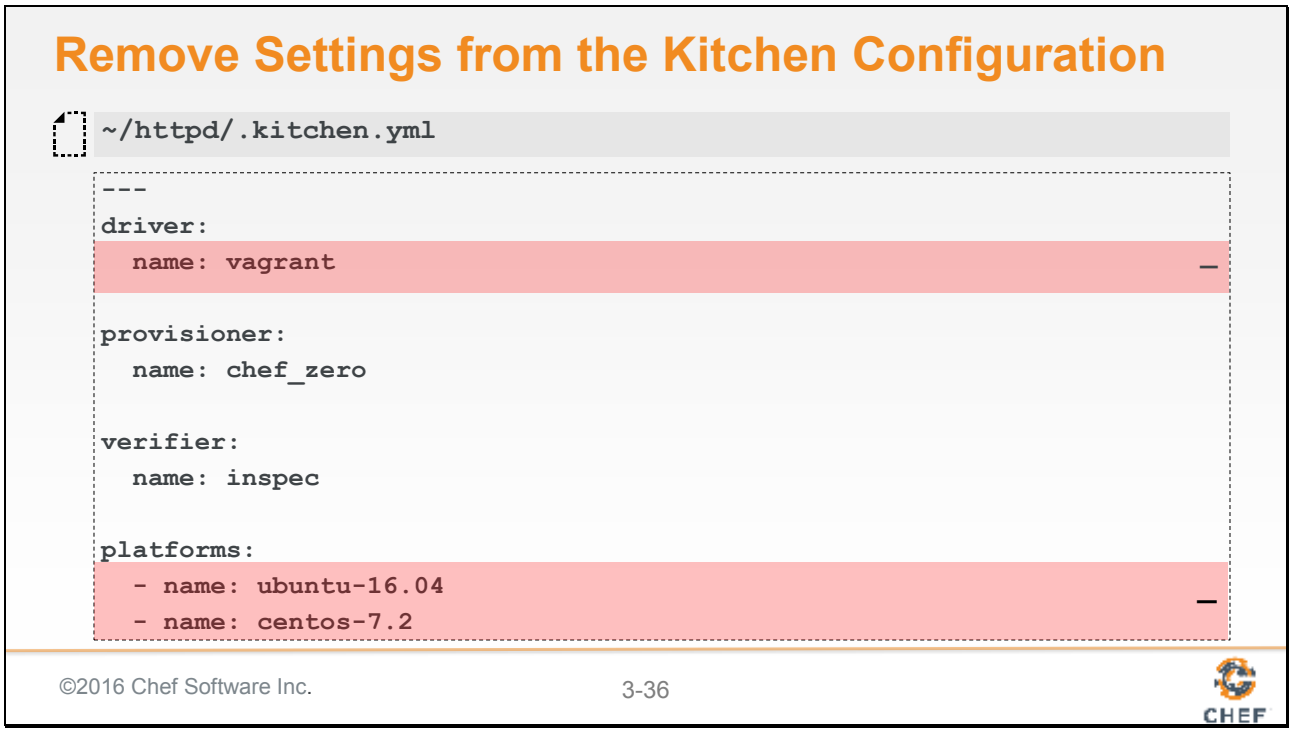
Slide 35

The Kitchen Suites' Tests

```
platforms:
  - name: ubuntu-16.04
  - name: centos-7.2

suites:
  - name: default
    run_list:
      - recipe[httpd::default]
    verifier:
      inspec_tests:
        - test/recipes
    attributes:
```

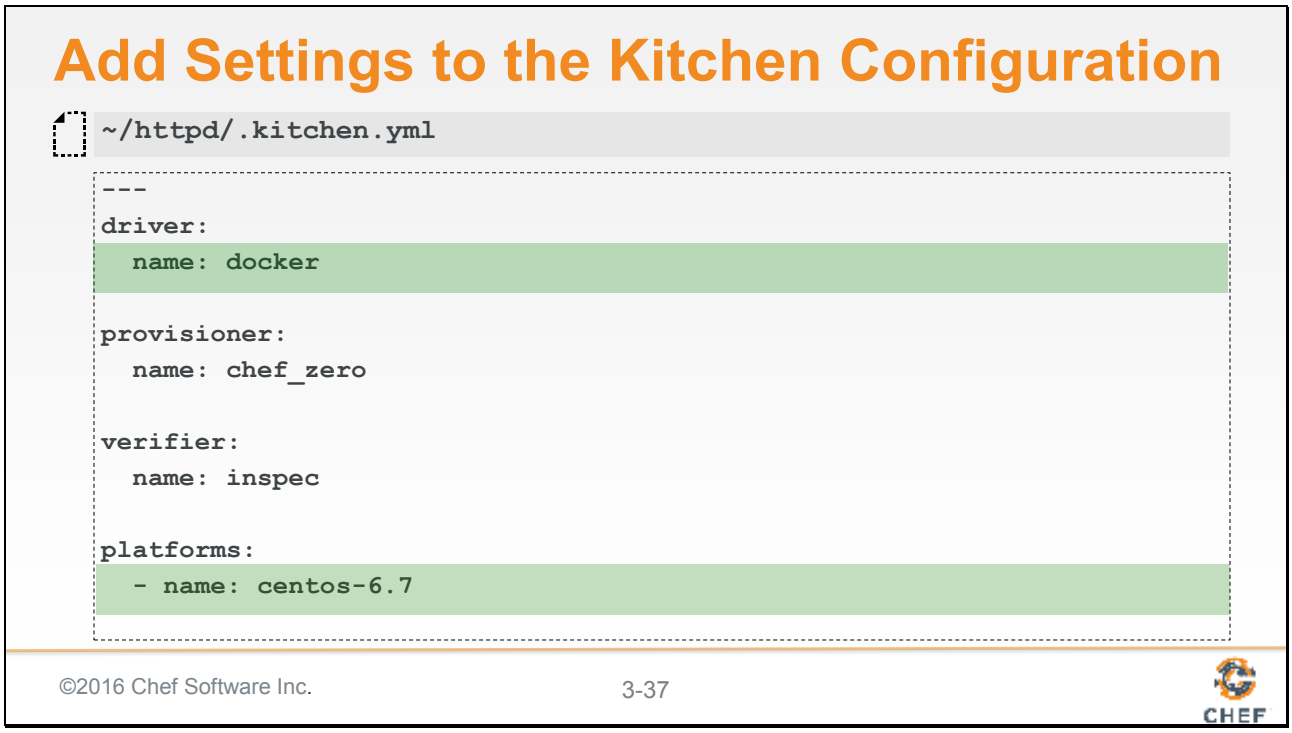
This is the path where the InSpec tests can be found.

This is the location where the tests can be found. This is the file that we viewed earlier and updated.



The initial Test Kitchen configuration is set up in way for local development on non-virtual machine. Because we are currently on a virtual machine we cannot use vagrant. We are also not interested in those following platforms.

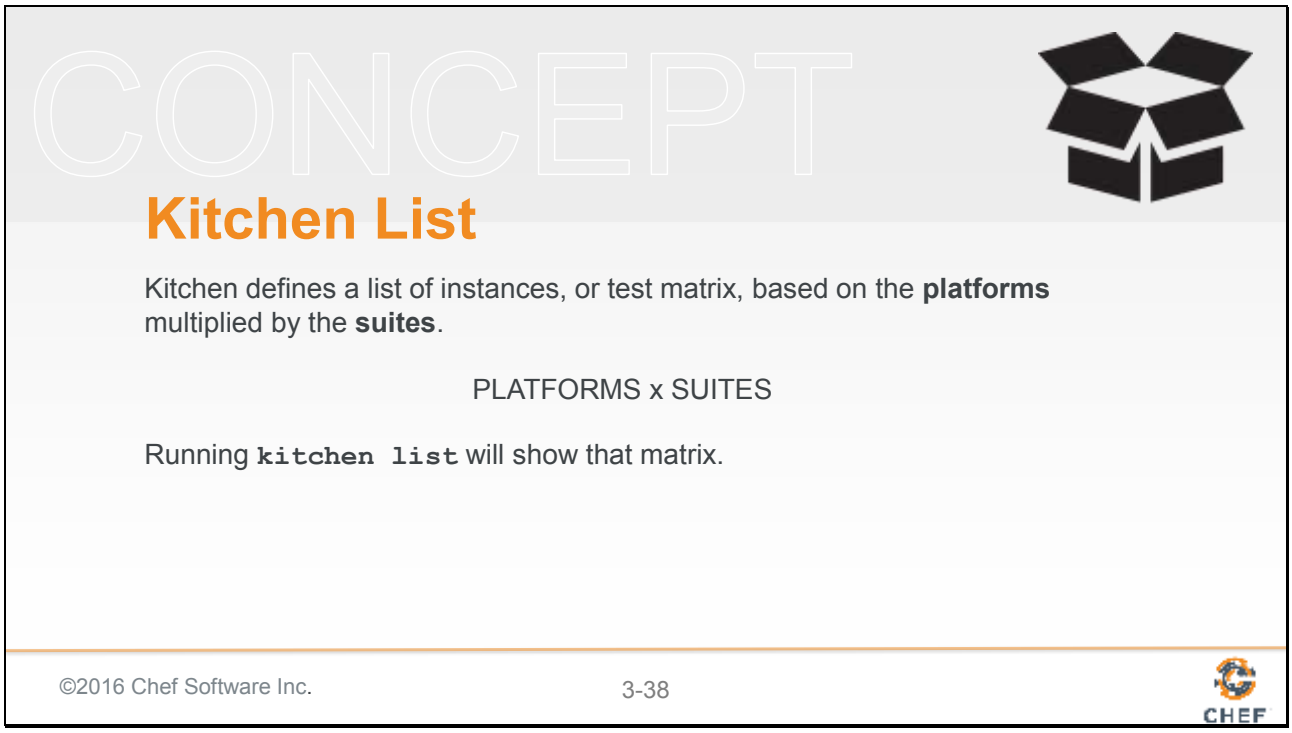
Slide 37



There are many different drivers that Test Kitchen supports. The docker driver is configured to work on this virtual machine. At this moment we are only interested in verifying that the cookbook we develop works on this current platform.

Later we will return to this configuration file and add an additional platform.

Slide 38



It is important to recognize that within the .kitchen.yml file we defined two fields that create a test matrix; the number of platforms we want to support multiplied by the number of test suites that we defined.

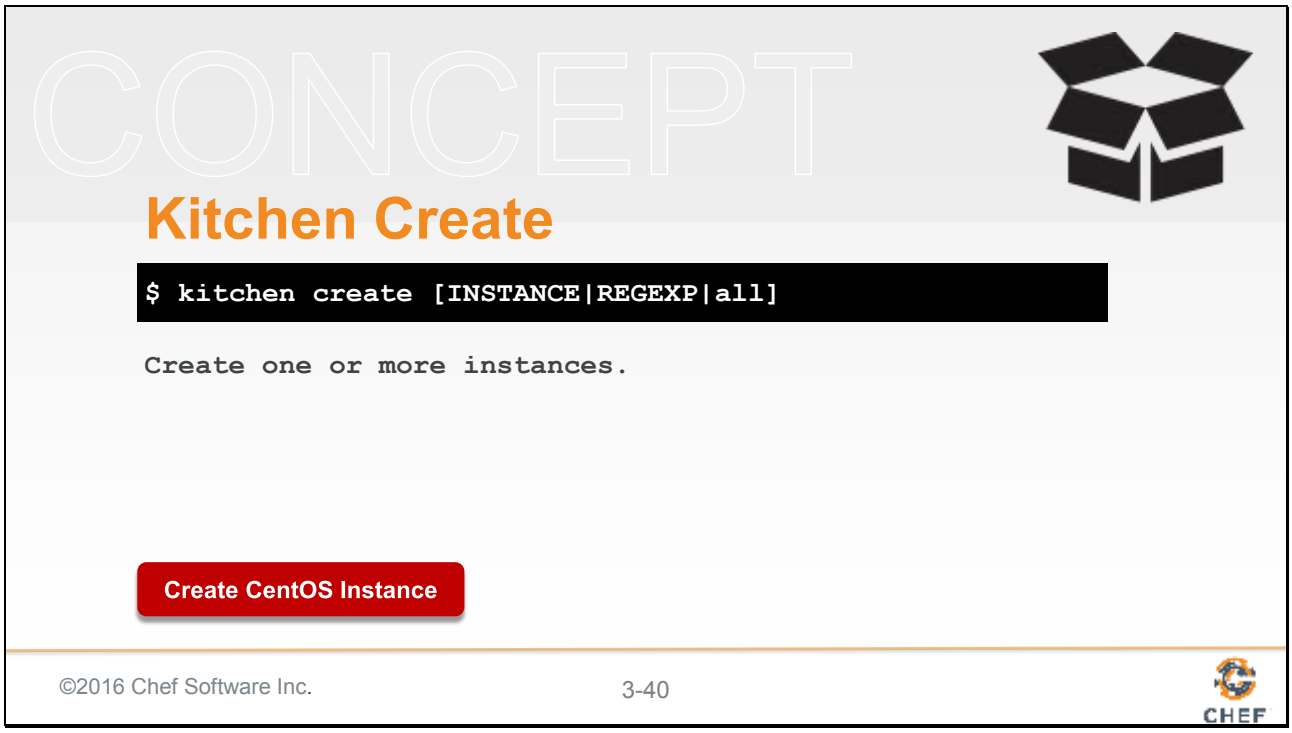
Slide 39

We can visualize this test matrix by running the command `kitchen list`.

In the output you can see that an instance is created in the list for every test suite and every platform. In our current file we have one suite, named 'default' and one platform CentOS.

Run the following command to verify that the Test Kitchen configuration file had been set up correctly.

Slide 40



Create or turn on a virtual or cloud instance for the platforms specified in the kitchen configuration.

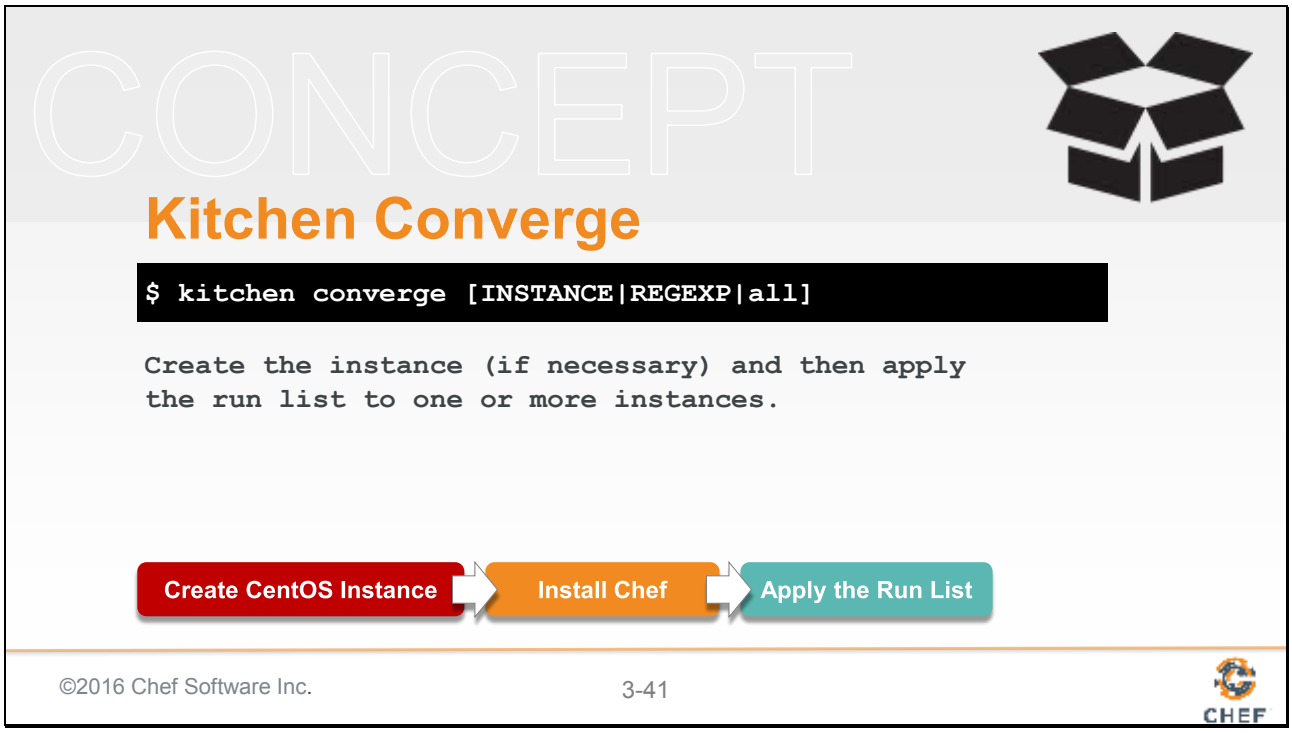
Running 'kitchen create default-centos-67' would create the the one instance that uses the test suite on the platform we want.

Typing in that name would be tiring if you had a lot of instances. A shortcut can be used to target the same system 'kitchen create default' or 'kitchen create centos' or even 'kitchen create 67'. This is an example of using the Regular Expression (REGEXP) to specify an instance.

When you want to target all of the instances you can run 'kitchen create' without any parameters. This will create all instances. Seeing as how there is only one instance this will work well.

In our case, this command would use the Docker driver to create a docker image based on centos-6.7.

Slide 41



Creating an image gives us a instance to test our cookbooks but it still would leave us with the work of installing chef and applying the cookbook defined in our .kitchen.yml run list.

So let's introduce you to the second kitchen command: 'kitchen converge'.

Converging an instance will create the instance if it has not already been created. Then it will install chef and apply that cookbook to that instance.

In our case, this command would take our image and install chef and apply the httpd cookbook's default recipe.

Slide 42

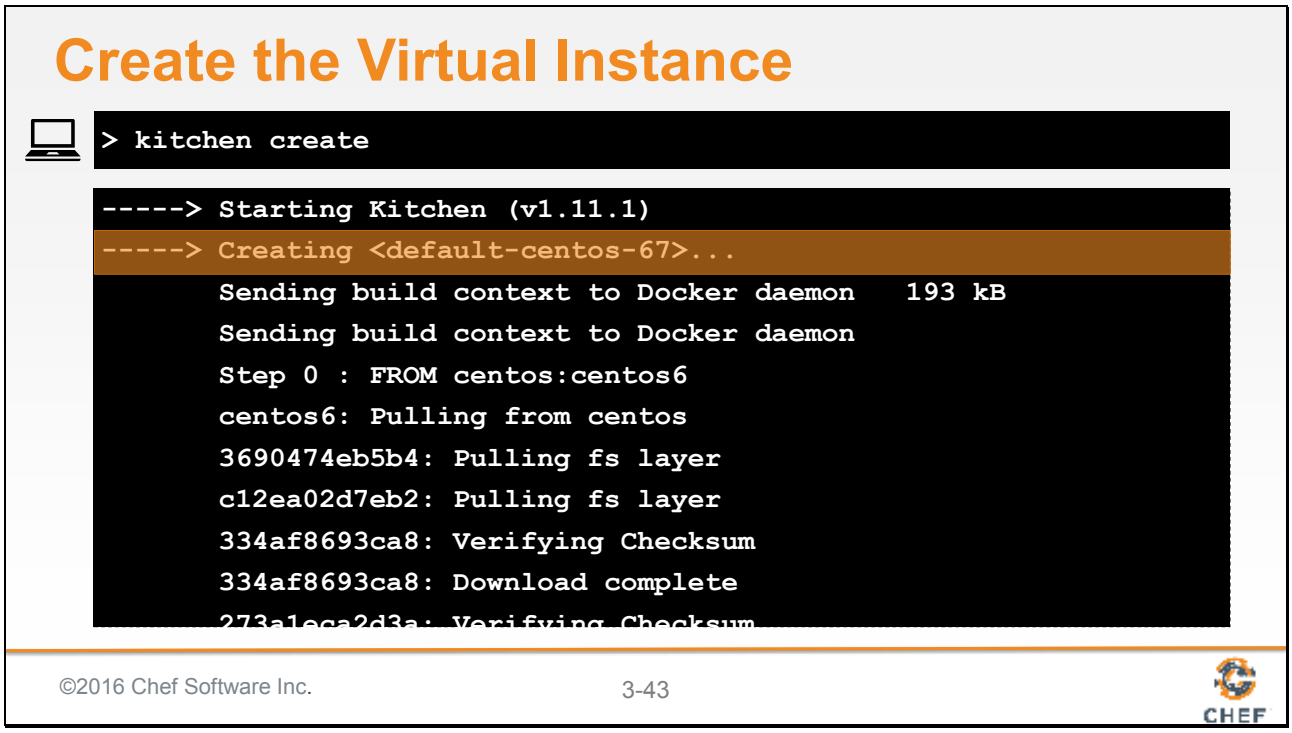
To verify an instance means to:

Create a virtual or cloud instances, if needed

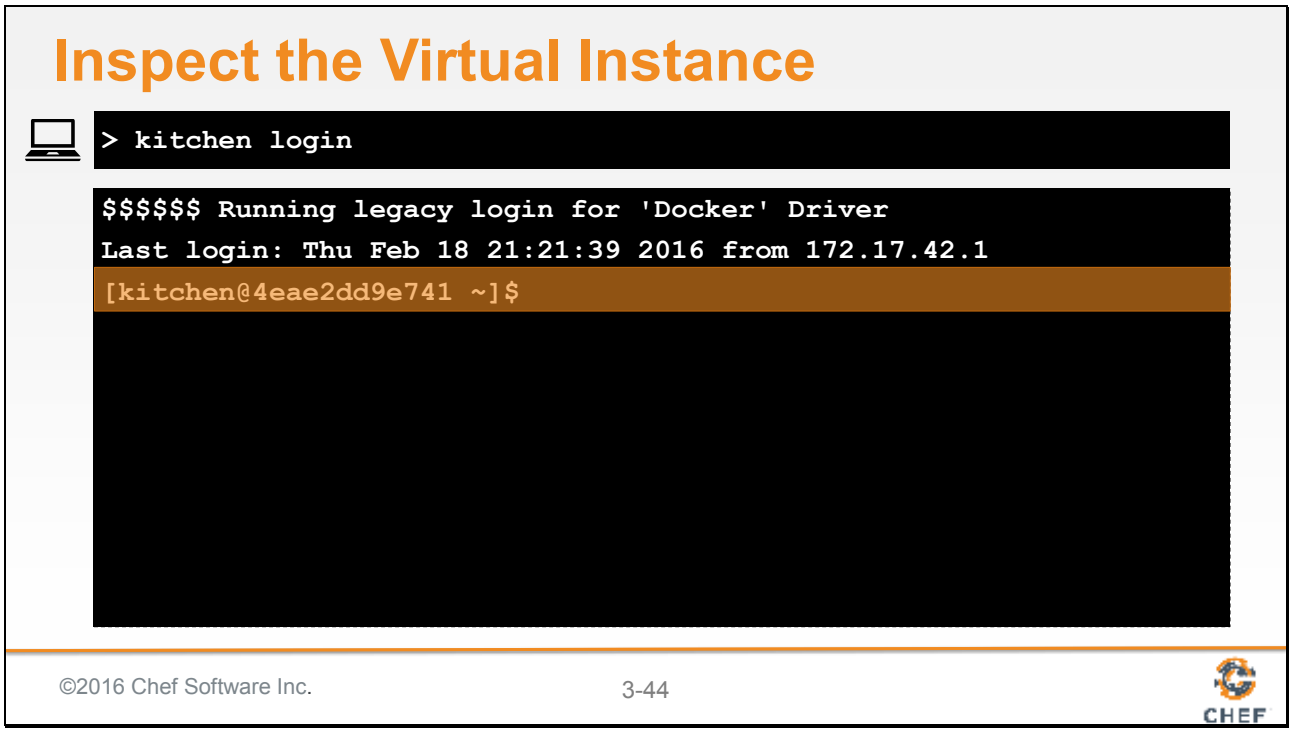
Converge the instance, if needed

And then execute a collection of defined tests against the instance

Slide 43



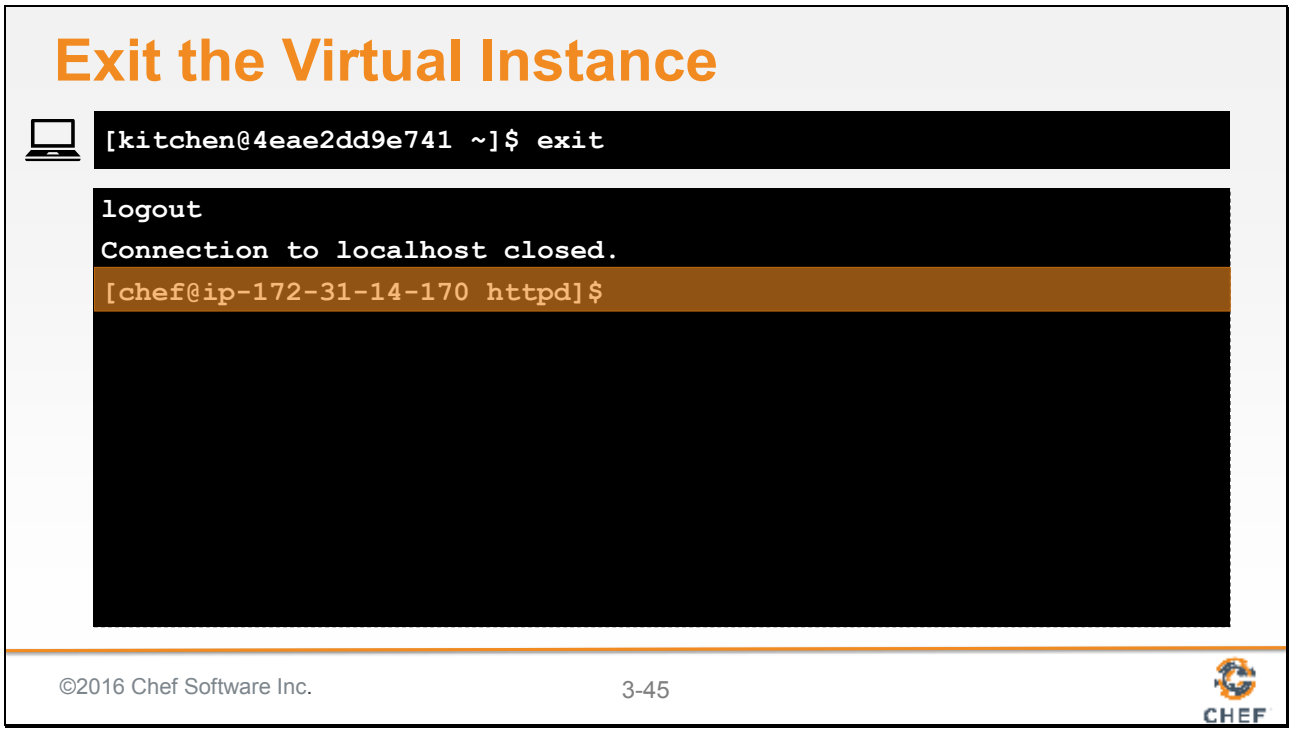
Create the instance with the following command. Here Test Kitchen will ask the driver specified in the kitchen configuration file to provision an instance for us.



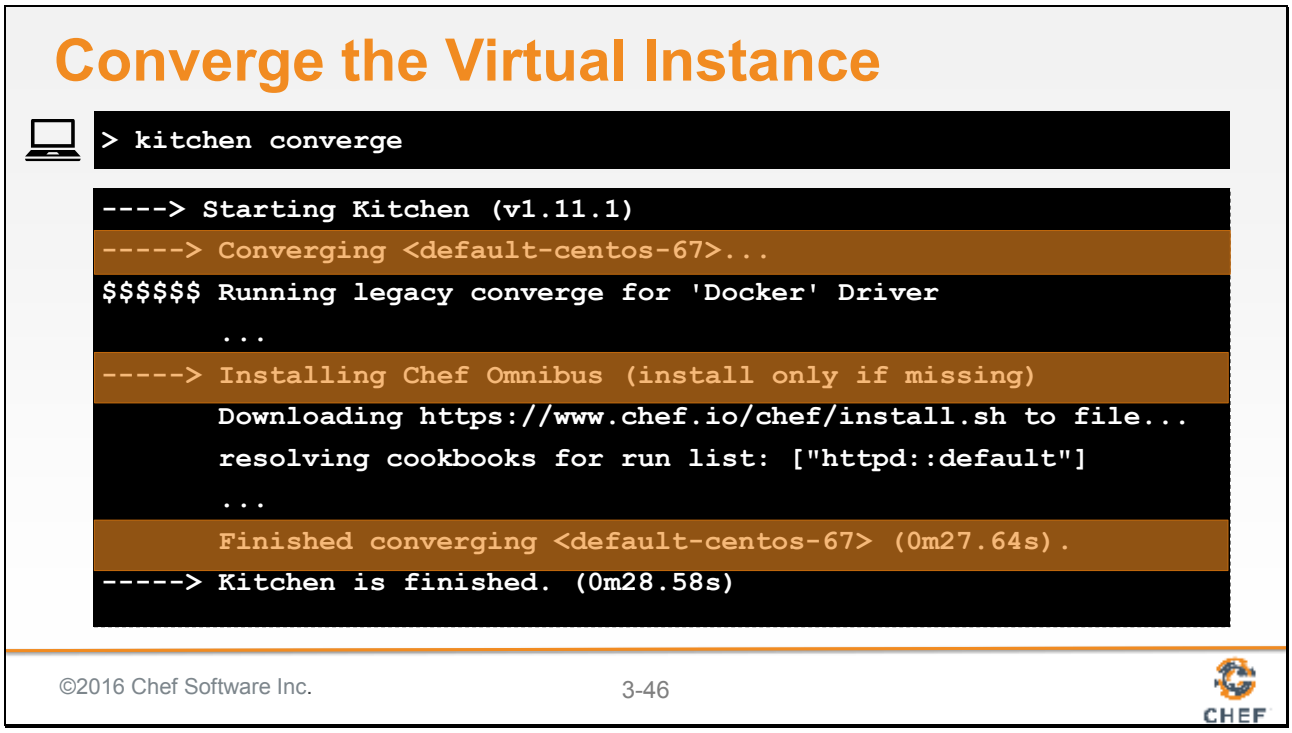
You can gain access to this virtual instance that we have created through the specified command. The login subcommand allows you to specify a parameter, which is the name of the instance that you want to log into. In your case, you only have one instance so Test Kitchen assumes you want to log into that one.

You are in now logged into a virtual instance on a virtual instance.

Slide 45



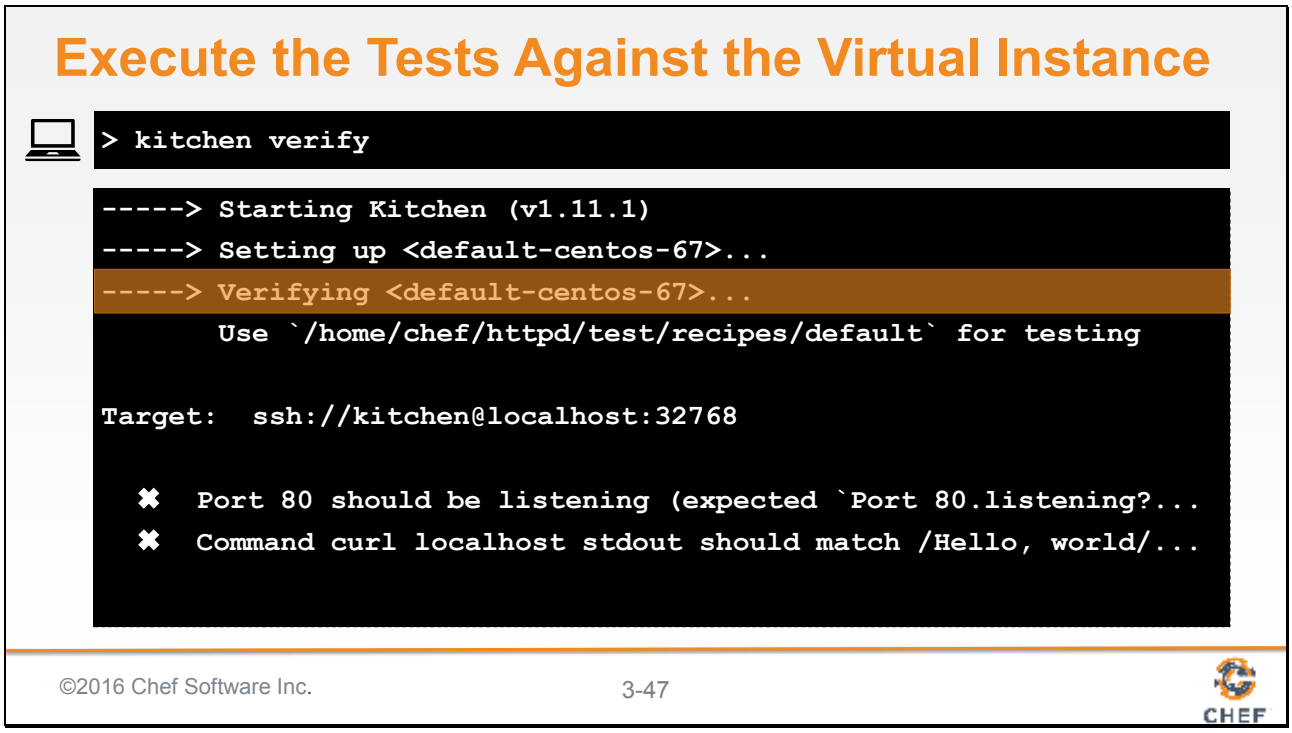
Logging in to the virtual instance is useful to explore the platform or assist with troubleshooting your recipes they fail in perplexing ways. Right now, we are interested in executing the tests so logout of the instance with the 'exit' command and we will return to the workstation.



Creating the instance allows us to view the operating system but Chef is not installed and the cookbook recipe, defined in the run list of the default test suite, has not been applied to the system. To do that you need to run 'kitchen converge'. Converge will take care of all of that.

In this instance the default recipe of the httpd cookbook contains no resources. You have not written a single resource that defines your desired state. Before we do that we want to ensure the instance is not already in a state that perhaps already meets the expectations that we defined.

Slide 47



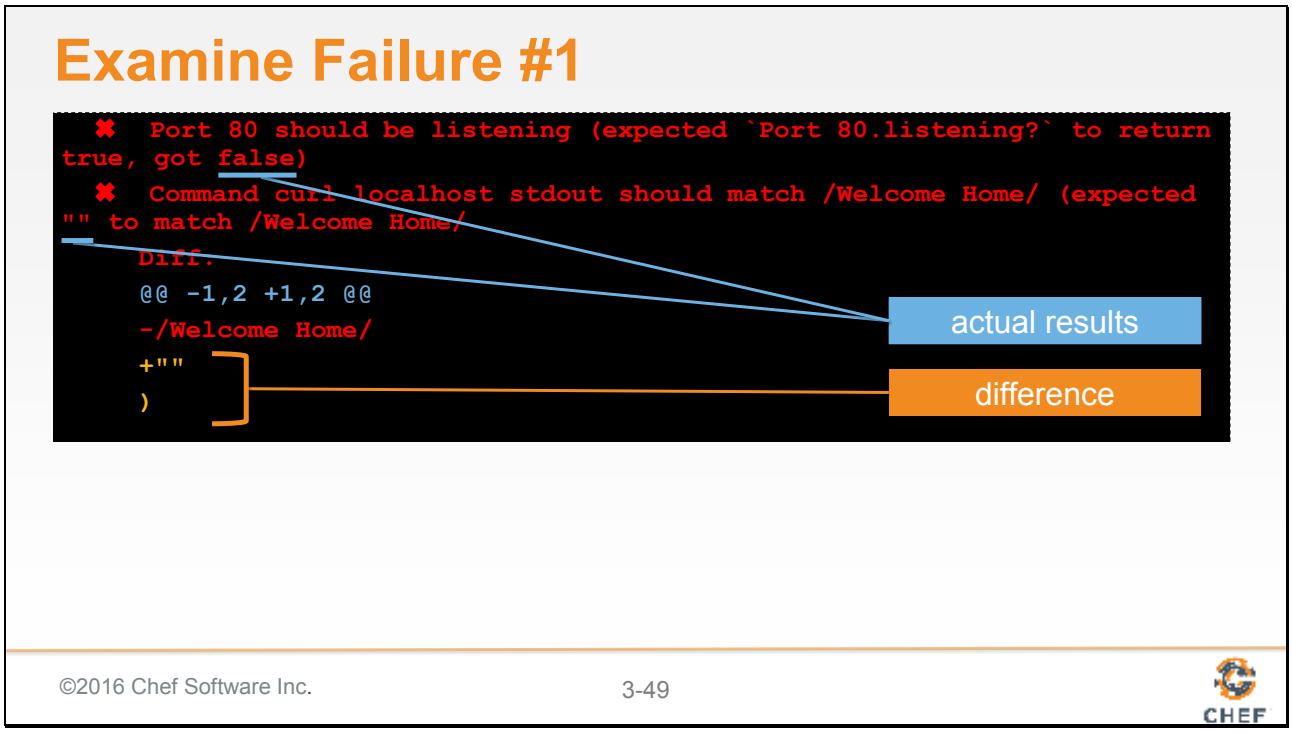
To verify the state of the instance with specification that we defined we use the 'kitchen verify' command. This command will install all the necessary testing tools, configure them, and then execute the test suite, and return to us the results.

Something that is important to mention is that we could have simply run this command from the start. When no previous instance exists, no instance has been created or converged, this command will automatically perform those two steps. When the instance is running, however, the verification step is only run.

Understanding the Failure Message

```
Target:  ssh://kitchen@localhost:32768

  ✖  Port 80 should be listening (expected `Port 80.listening?` to return true, got false)
  ✖  Command curl localhost stdout should match /Welcome Home/ (expected "" to match /Welcome
Home/
     Diff:
     @@ -1,2 +1,2 @@
     -/Welcome Home/
     +""
     )

Summary: 0 successful, 2 failures, 0 skipped
>>>>>> ------Exception-------
>>>>>> Class: Kitchen::ActionFailed
>>>>>> Message: 1 actions failed.
>>>>>>     Verify failed on instance <default-centos-67>.  Please see .kitchen/logs/defau...
```


Now, let's read the results from the kitchen verification to ensure that our expectations failed to be met.



Each failure is displayed with a human-readable sentence about the defined resource, the expected results, and the result that was received (or 'got').

We see that we have two errors. The first is that port 80 is not listening when we expected to be listening. We also expected the command's standard out to return content to us and it did not; it returned nothing.

Slide 50

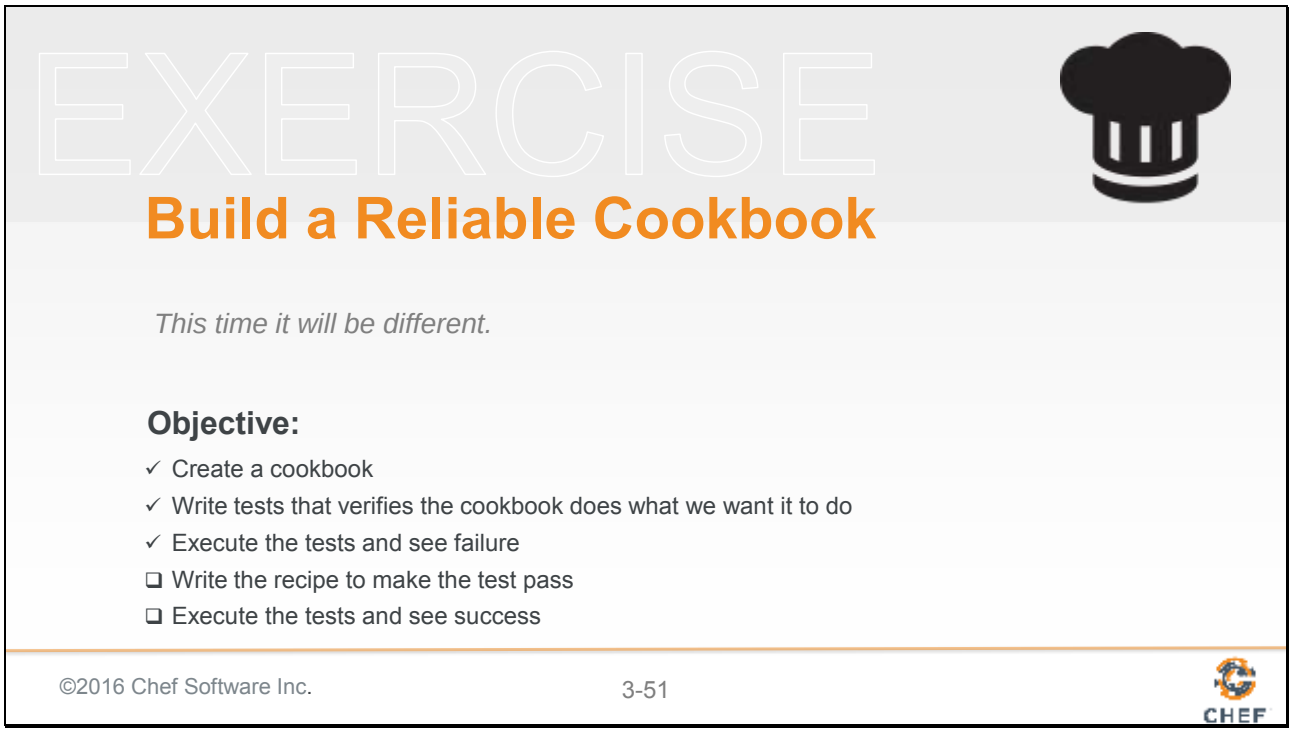
Examine the Test Summary

```
  ✖  Port 80 should be listening (expected `Port 80.listening?` to return true, got false)
  ✖  Command curl localhost stdout should match /Welcome Home/ (expected "" to match /Welcome
Home/
     Diff:
     @@ -1,2 +1,2 @@
     -/Welcome Home/
     +""
     )

Summary: 0 successful, 2 failures, 0 skipped
```

A final summary contains the length of execution time with the results shows that RSpec verified 2 examples and found 2 failures.

After all the failures a final summary of the results will be displayed which shows us that our test suite contains 2 examples and that 2 examples failed to meet expectations.

Slide 51



Now we know for certain that the test instance is not in our desired state. When we write the resources now in the default recipe to bring the instance to the desired state we can be certain that we have done it in a way that meets the expectations that we have established.

Slide 52

Write the Default Recipe for the Cookbook

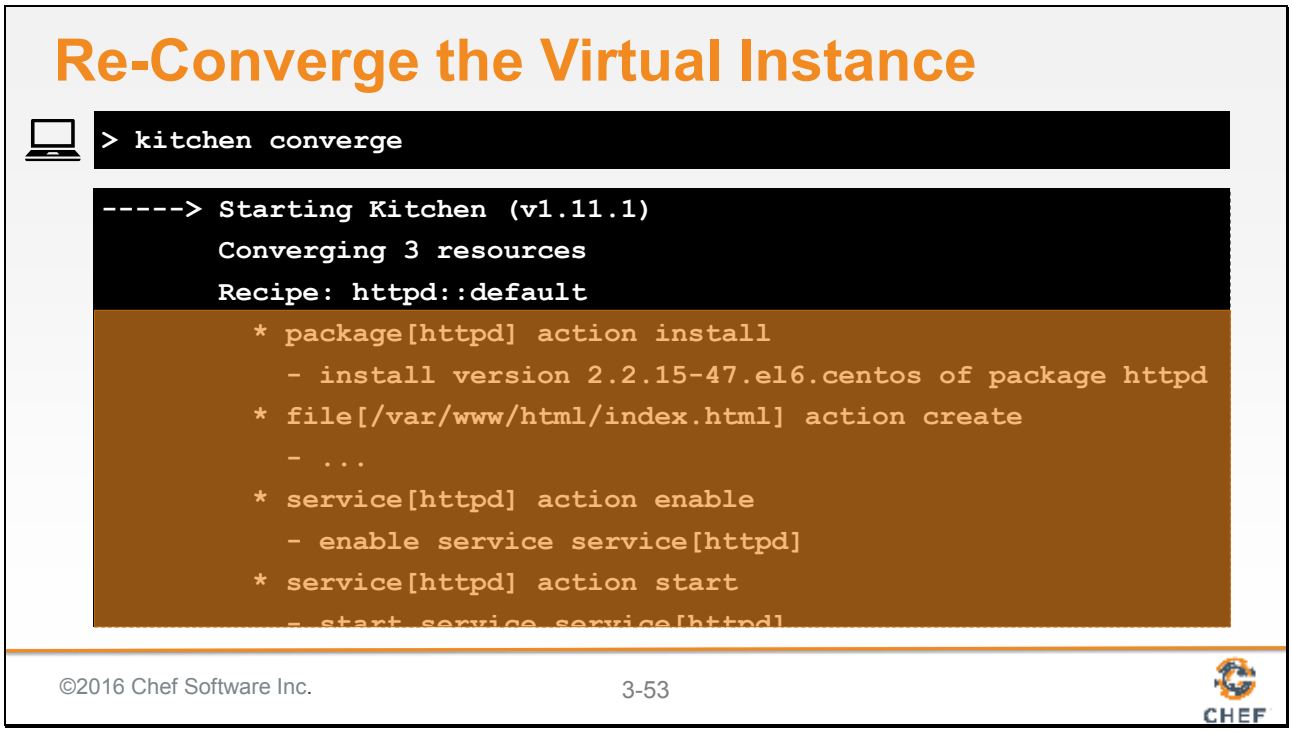
~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Welcome Home!</h1>'
end

service 'httpd' do
  action [:enable, :start]
end
```


The following recipe defines three resources. These three resources express the desired state of an apache server that will serve up a simple page that contains the text 'Welcome Home!'.

The package will install all the necessary software on the operating system. The file will create an HTML file with the desired content at a location pre-defined by the web server. The service resource will start the web server and then ensure that if we reboot the system the web server will start up.

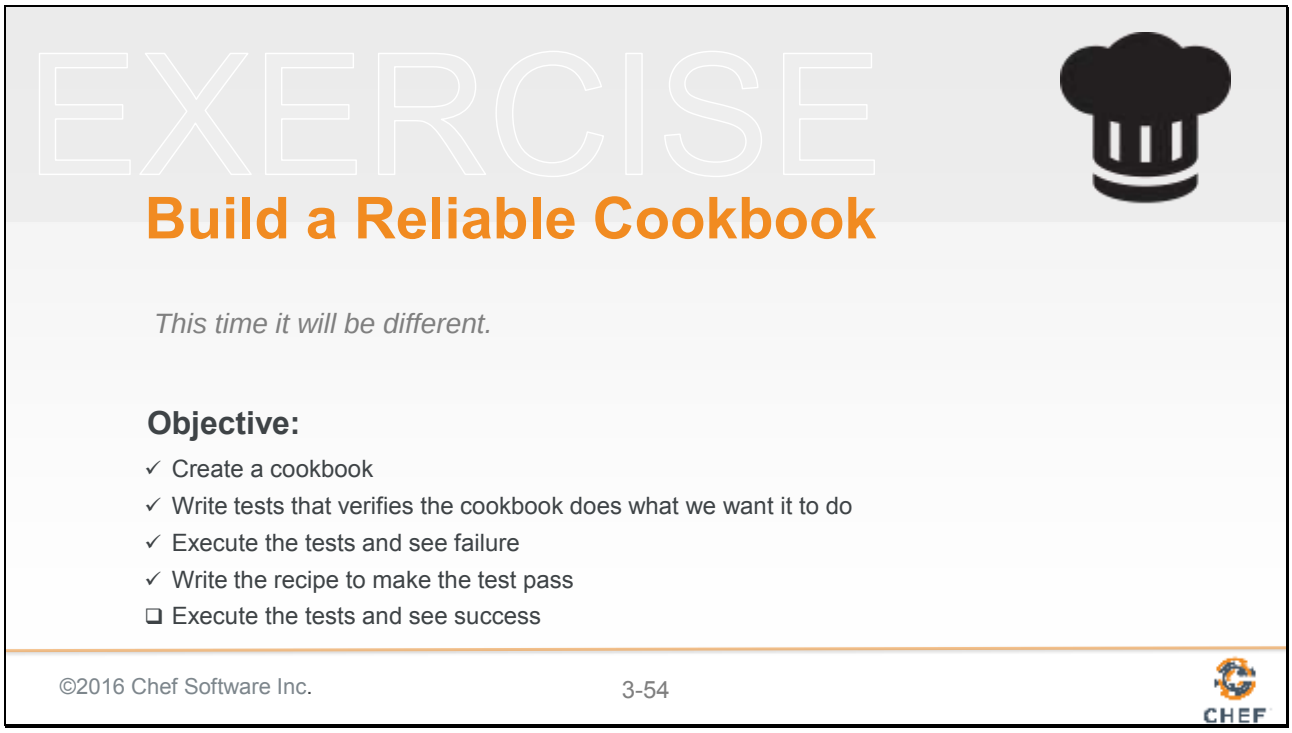
Slide 53



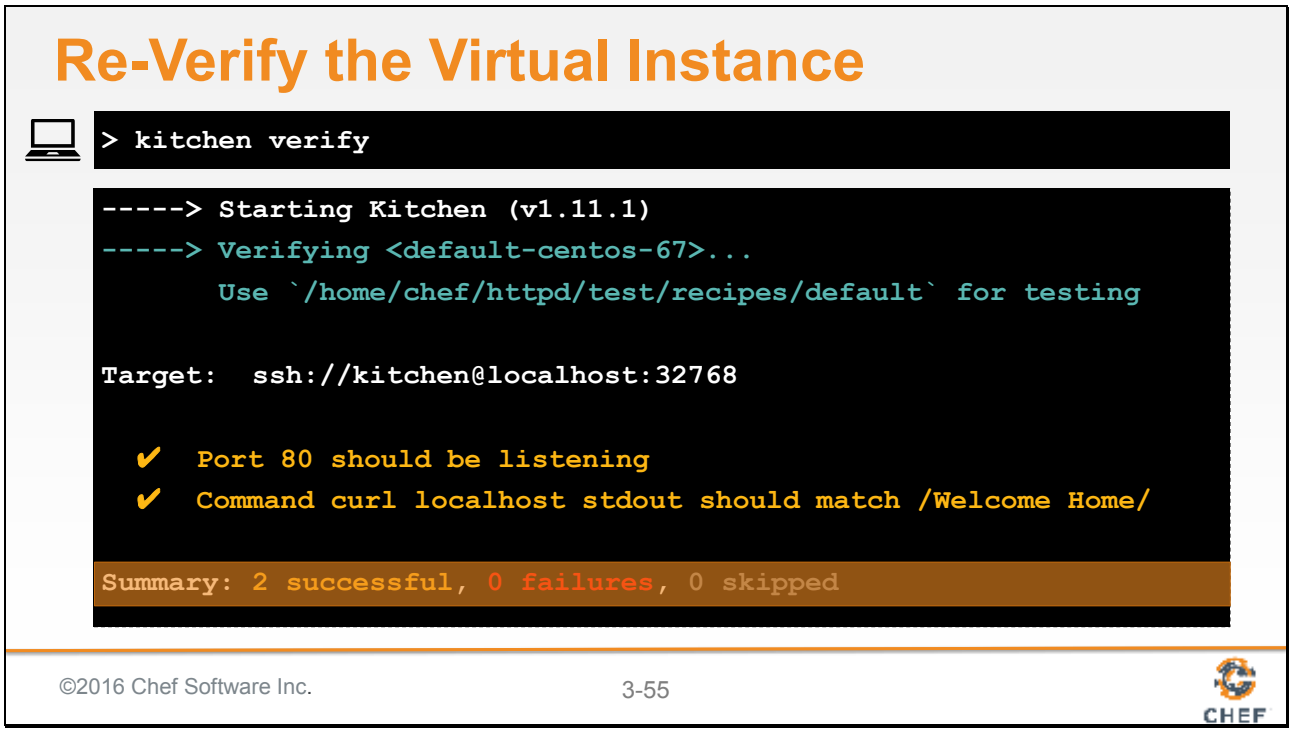
Whenever you make a change to the recipe it is important to run 'kitchen converge'. This command will apply the updated recipe to the state of the virtual instance.

In the output, you should see the resources that you defined being applied to the instance. The package, the file, and the actions of the service.

Slide 54



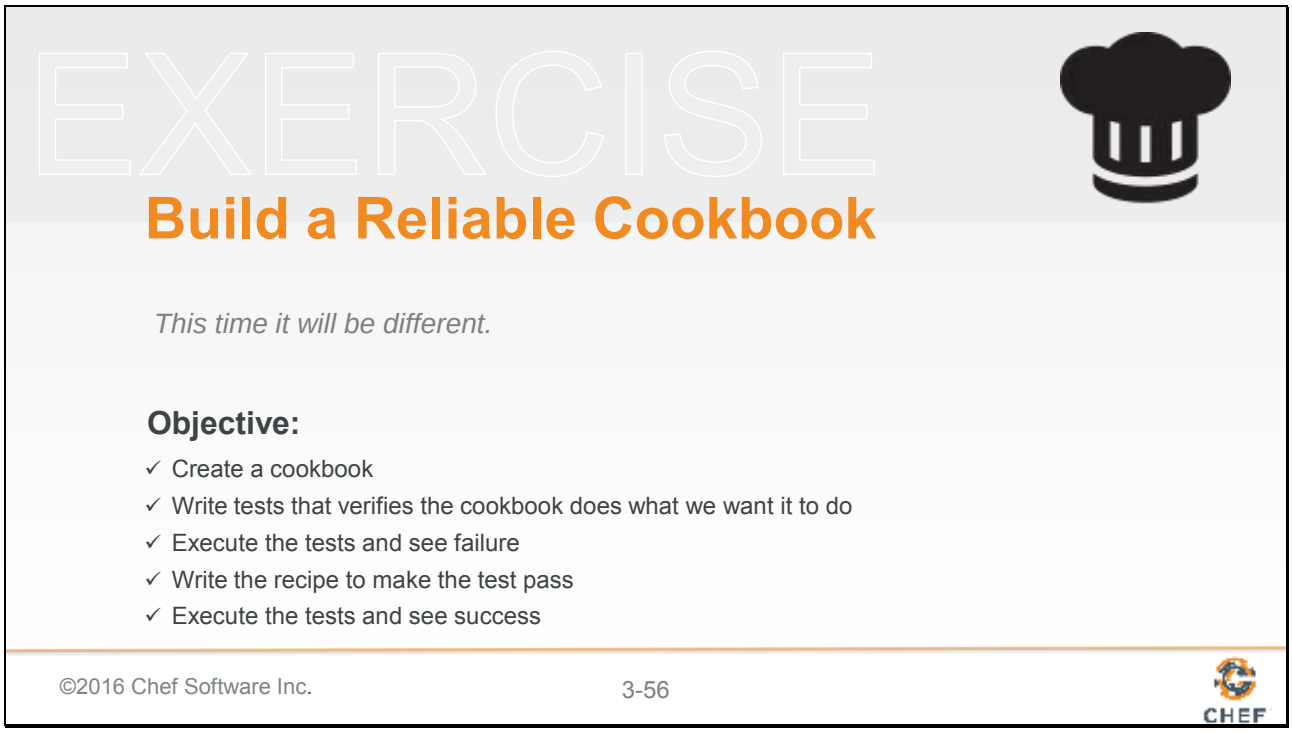
Now with the desired state expressed in the default recipe and applied to the virtual instance it is time to see if the test we wrote initially will now pass. If it does, that means we got everything right in the configuration we wrote in the recipe. We can declare victory!



To verify the state of the virtual instance you run the 'kitchen verify' command. In the summary you should find the failing expectation no longer fails.

If it does fail, it is time to review the code you wrote in the recipe file and the spec file. When it was failing did you get a different failure than the one that we walked through? That probably means there is an error in the spec file. Did the test instance actually converge successfully? Sometimes output will scroll by and we don't have time to read it. I get it. Scroll back up and see if there was an error message tucked into the 'kitchen converge' you ran.

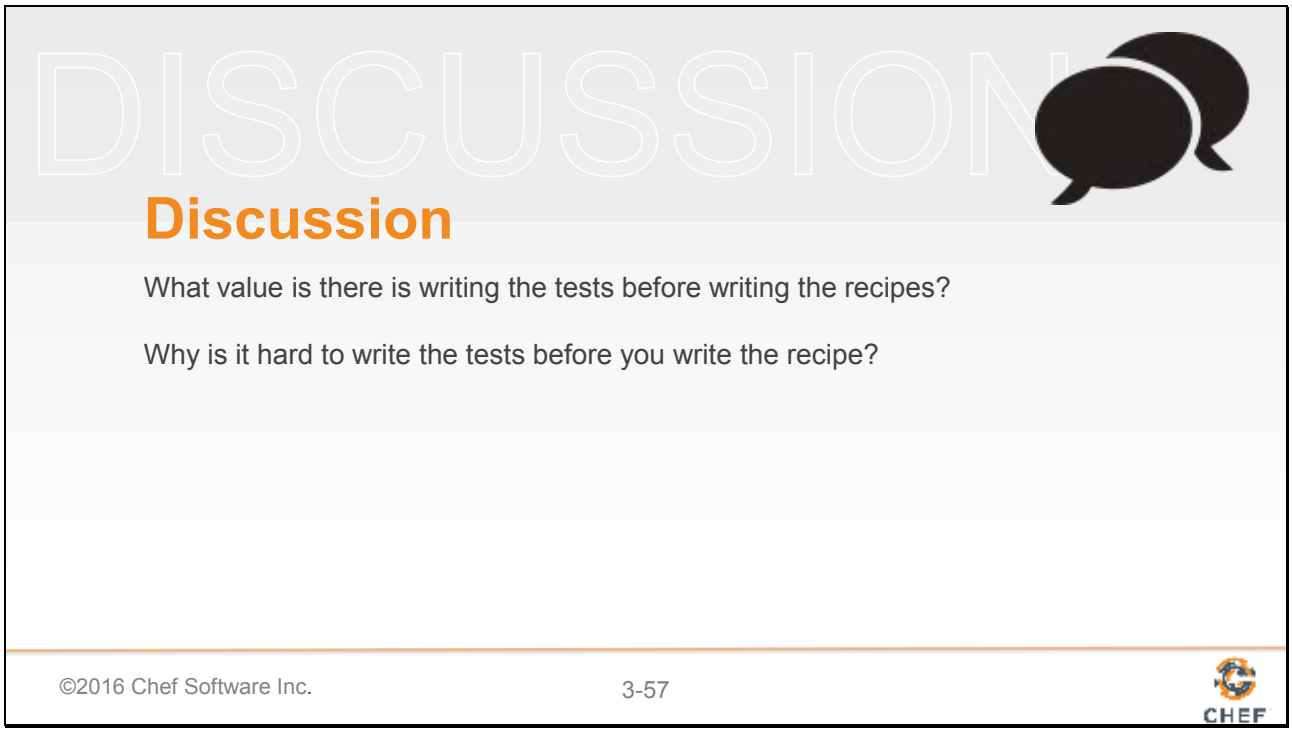
Slide 56



So you've done it. You have done Test Driven Development (TDD). Wrote a test. Saw it fail. Wrote a unit of code. Saw it pass.

You created a cookbook. Wrote an expectation in the spec file. Saw the test fail. Wrote a recipe. Applied the recipe. Ran the tests and saw them pass.

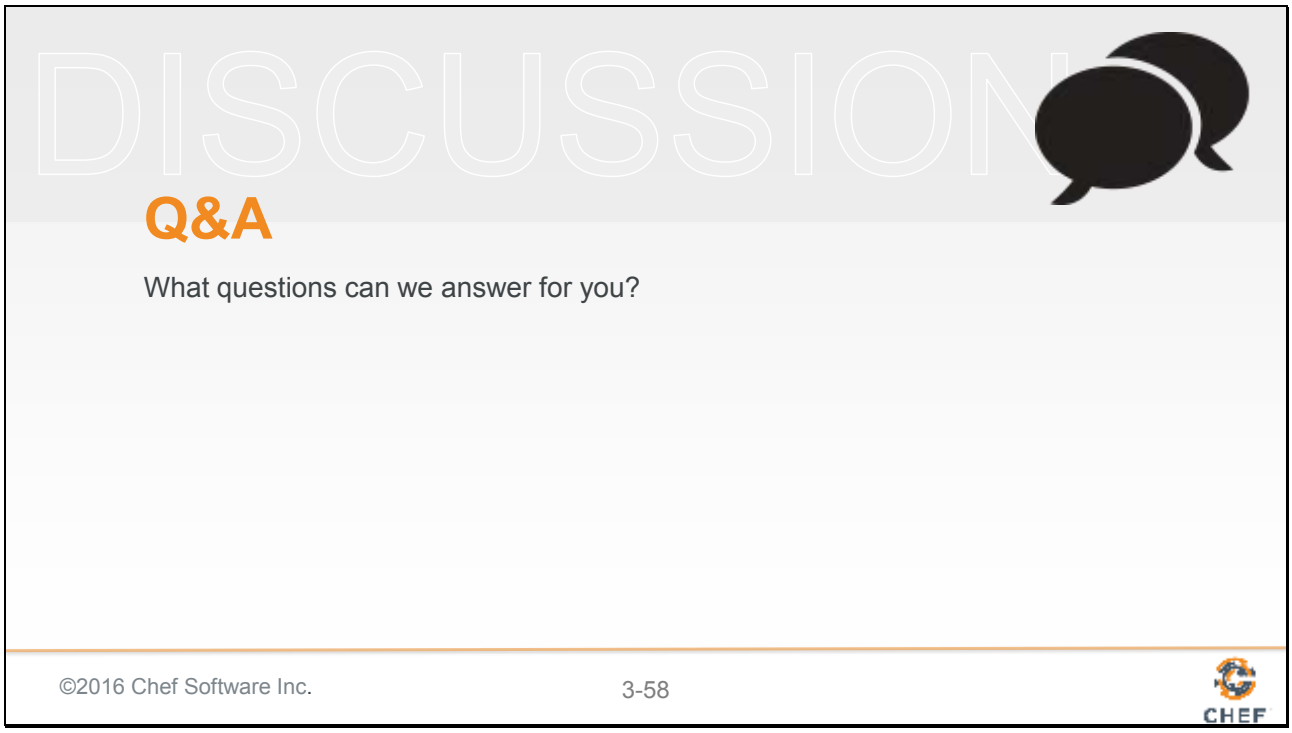
Slide 57



Now that you participated in writing a test and then the recipe let's have a discussion.

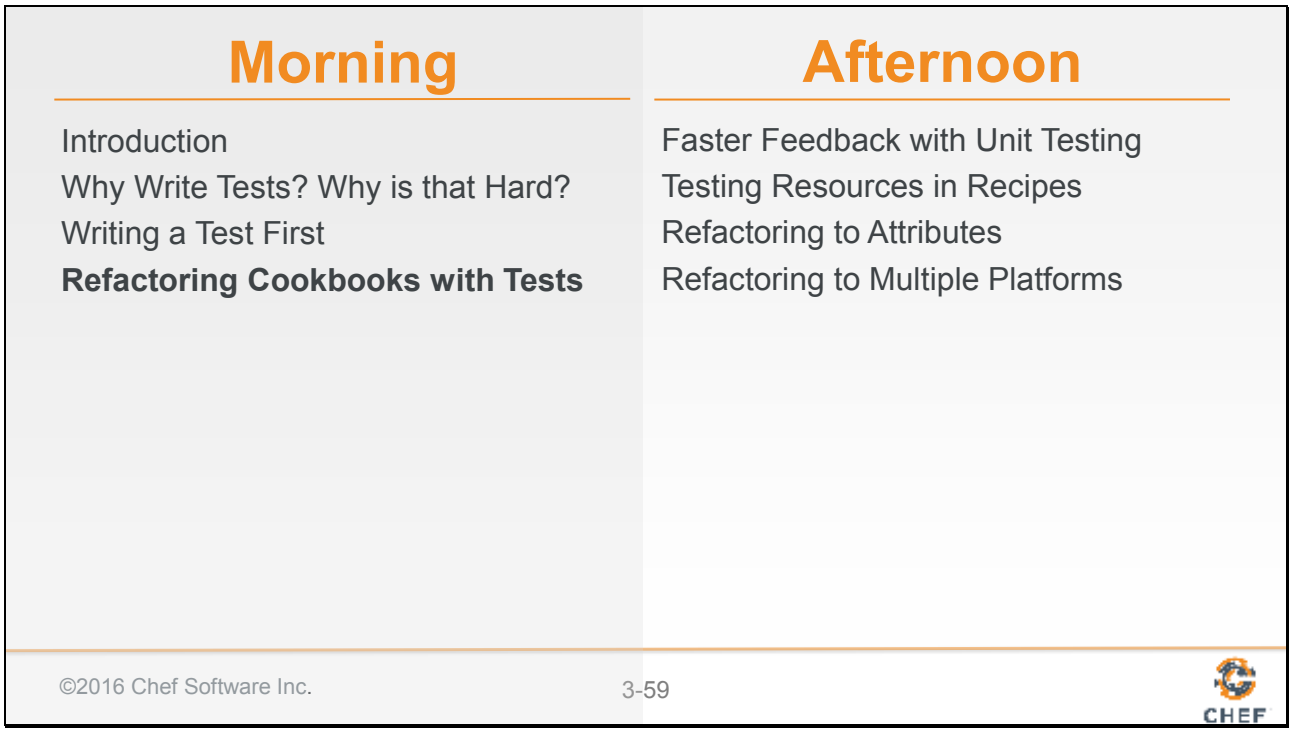
Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 58



Before we complete this section, let us pause for questions.

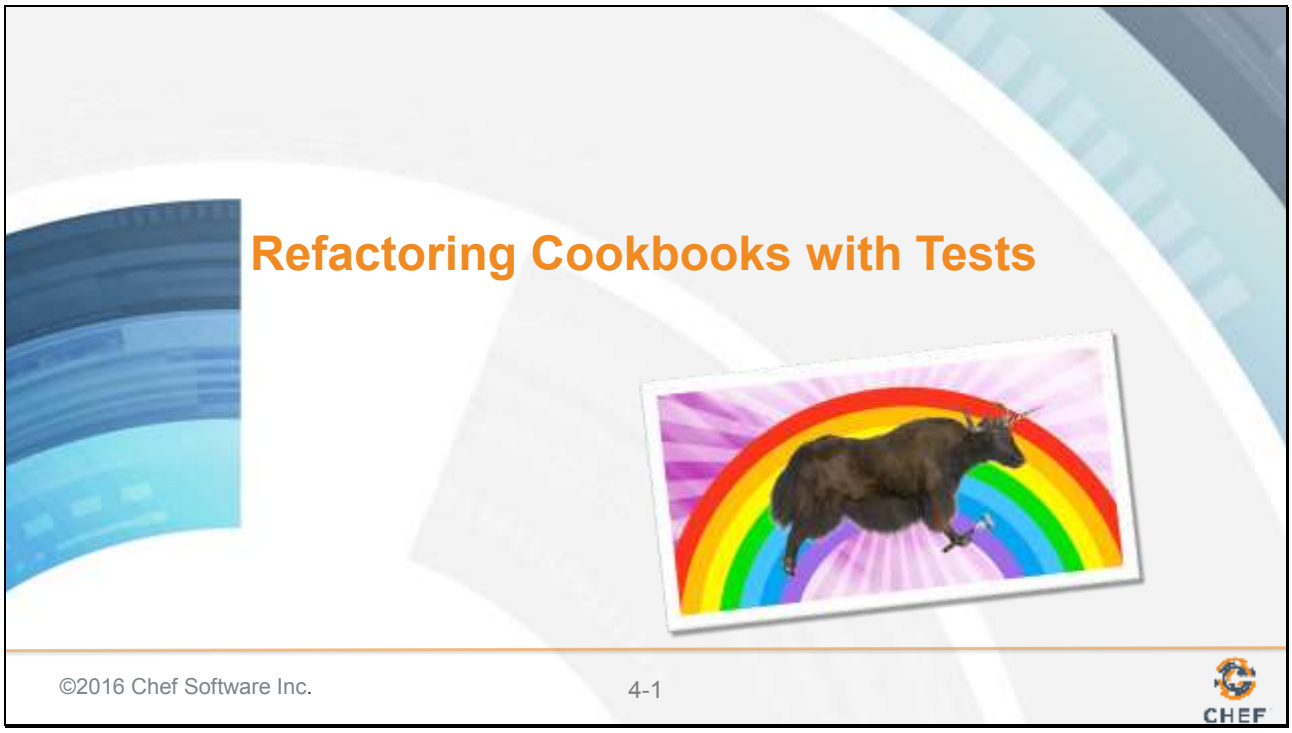
Slide 59



You have performed almost all of the steps of TDD. Next we are going to use the tests to help us refactor the recipe we wrote. In a series of group exercises we will explore some of the important nuances of Test Kitchen's subcommands: converge and verify. And explore another subcommand named: test.

Slide 60

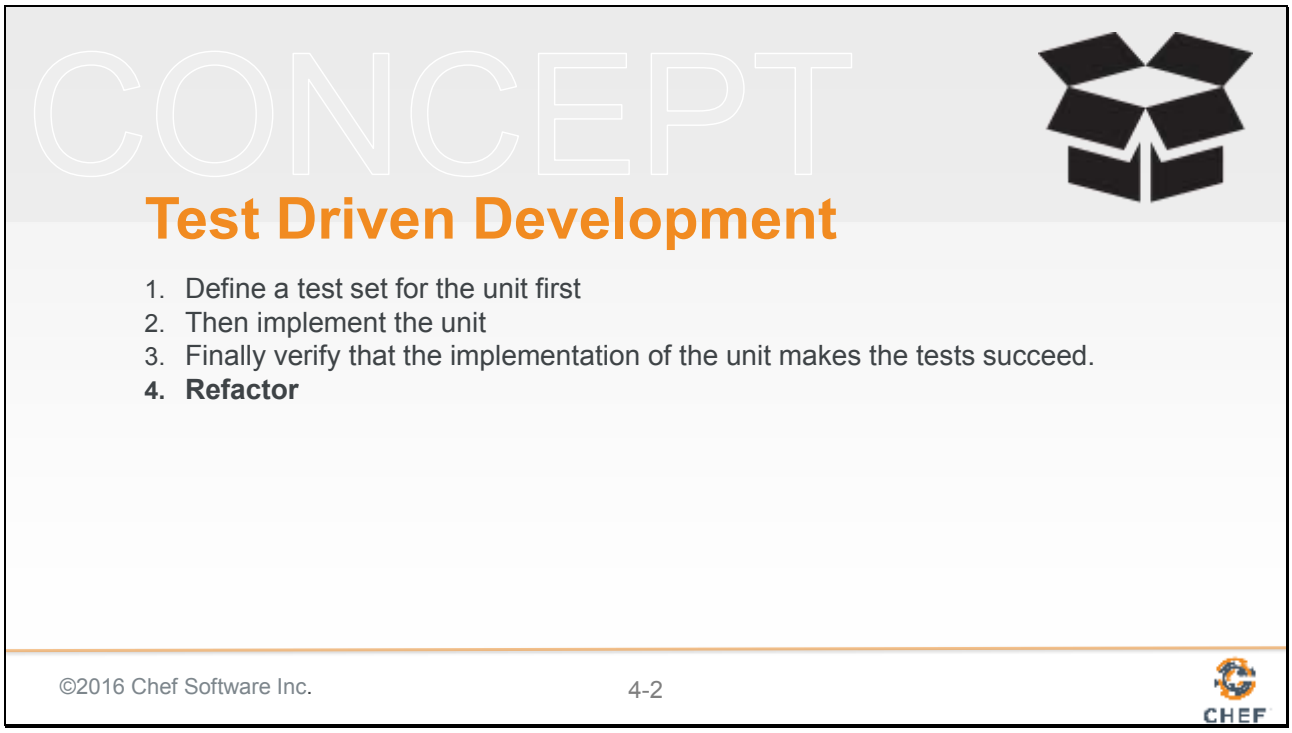
4: Refactoring Cookbooks with Tests



We explored the process of developing a test first but to explore the full Test Driven Development (TDD) cycle we need to refactor the code that we wrote.

Refactoring is the process of making changes to the implementation while maintaining the original intention. Without having tests that capture the original intention how do you know if the new implementation did not change the original intention? Fortunately for us we have defined a test that will allow us to make the changes confident that we have not destroyed that original intention.

Slide 2



Refactoring is the often forgotten step in the TDD cycle. When we are able to get our expectations to pass we immediately want to move to our next requirement or next cookbook.

This step is incredibly important. Within it we are able to reflect on the unit of code and tests that we have written and evaluate them. How you evaluate the code may vary based on your experience, the standards defined by the team you work with, or if the code will be shared with the Chef community.

Slide 3

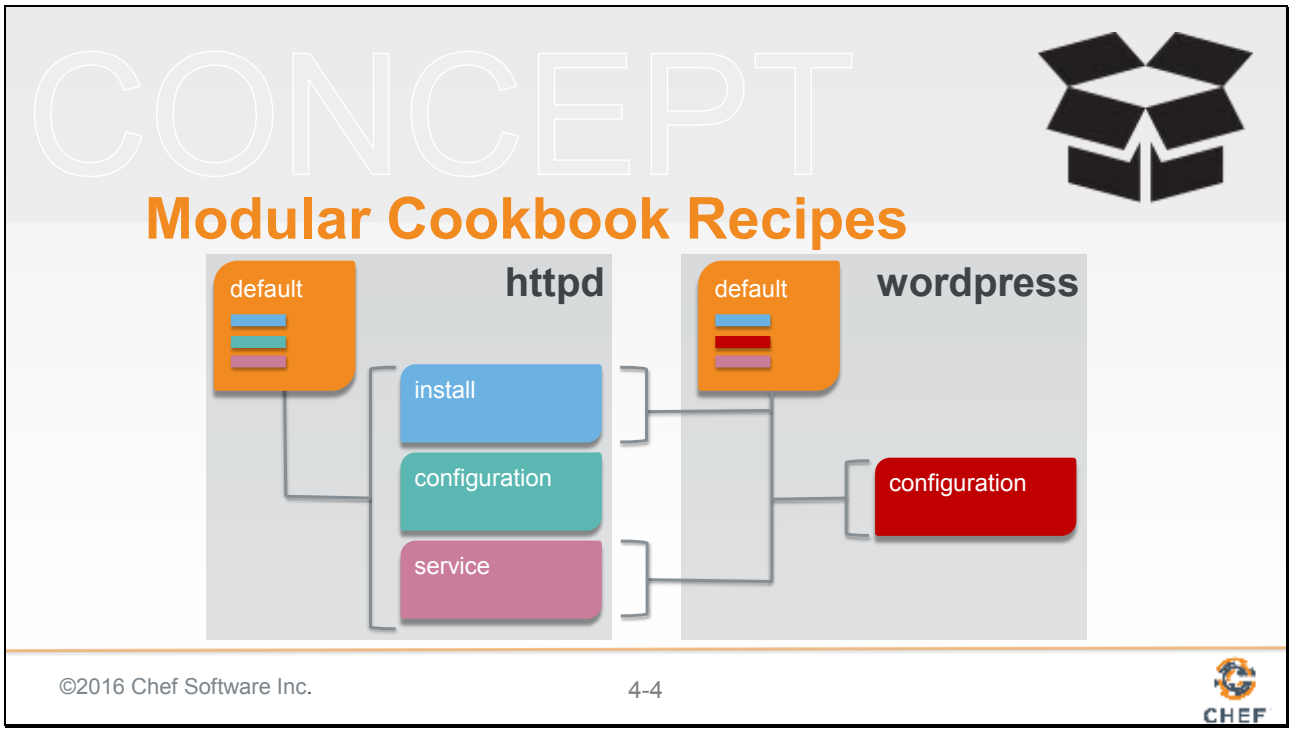
Objectives

After completing this module, you should be able to:

- Refactor a recipe using `include_recipe`
- Use Test Kitchen to validate the code you refactored
- Explain when to use `kitchen converge`, `kitchen verify` and `kitchen test`.

In this module you will learn how to refactor a cookbook using the method 'include_recipe', verify the changes with Test Kitchen, and then explain in what scenarios you would choose to use 'kitchen converge', 'kitchen verify' and 'kitchen test'.

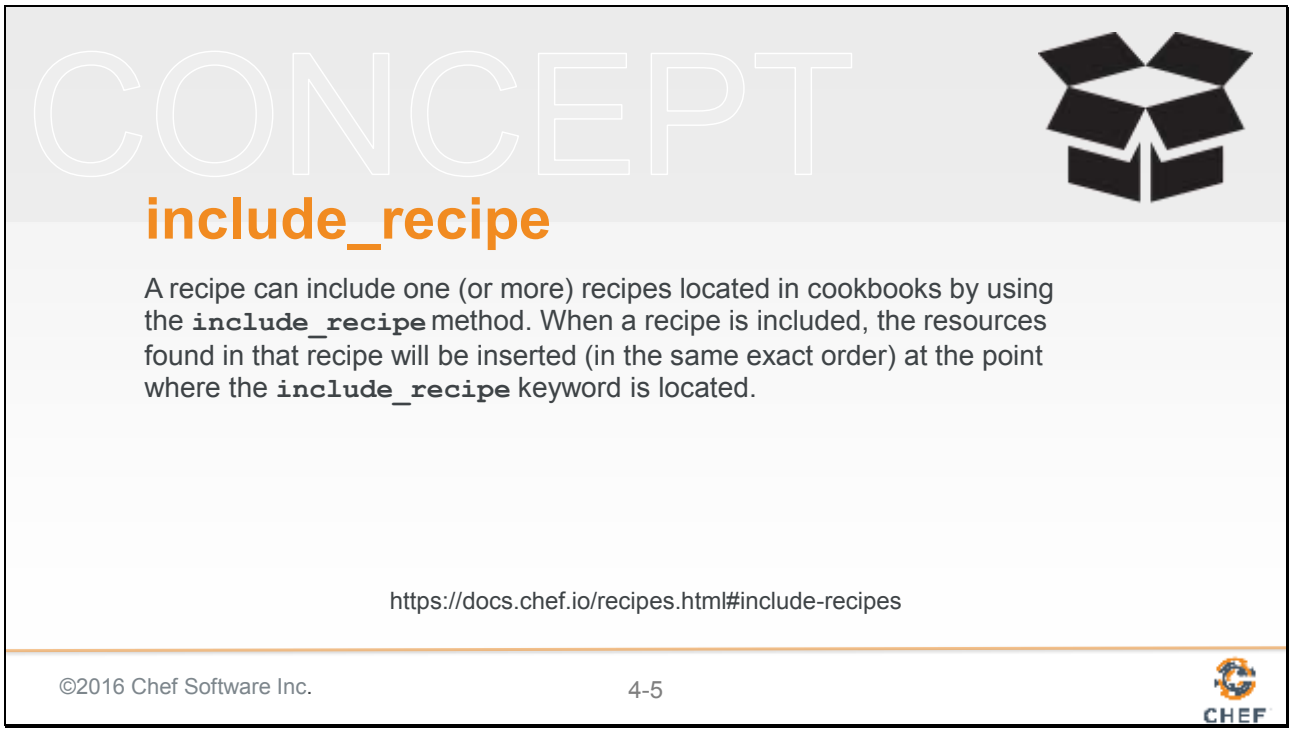
Slide 4



Our initial implementation of the default recipe for the httpd cookbook defined the entire installation, configuration, and management of the service within a single recipe. This implementation has the benefit of being entirely readable from a single recipe. However, it does not easily allow for other cookbooks that may want to use the httpd cookbook to easily choose the components that it may need.

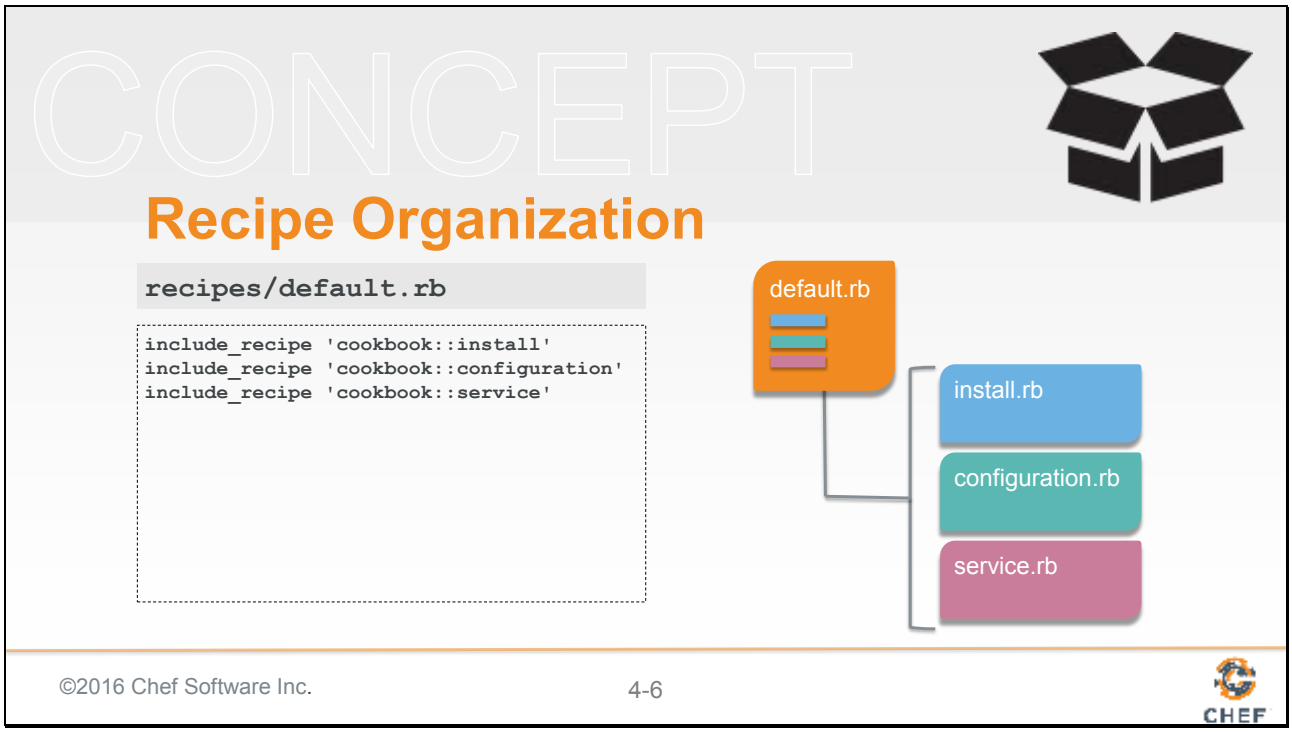
An example of this is that we may deploy wordpress or some other web application that relies on the apache webserver installed and running. In this new cookbook we would like to re-use the resources that installs apache and the resources that manage the service. We most likely do not want to setup a test page that greets people. We are likely going to replace it with application code.

Slide 5



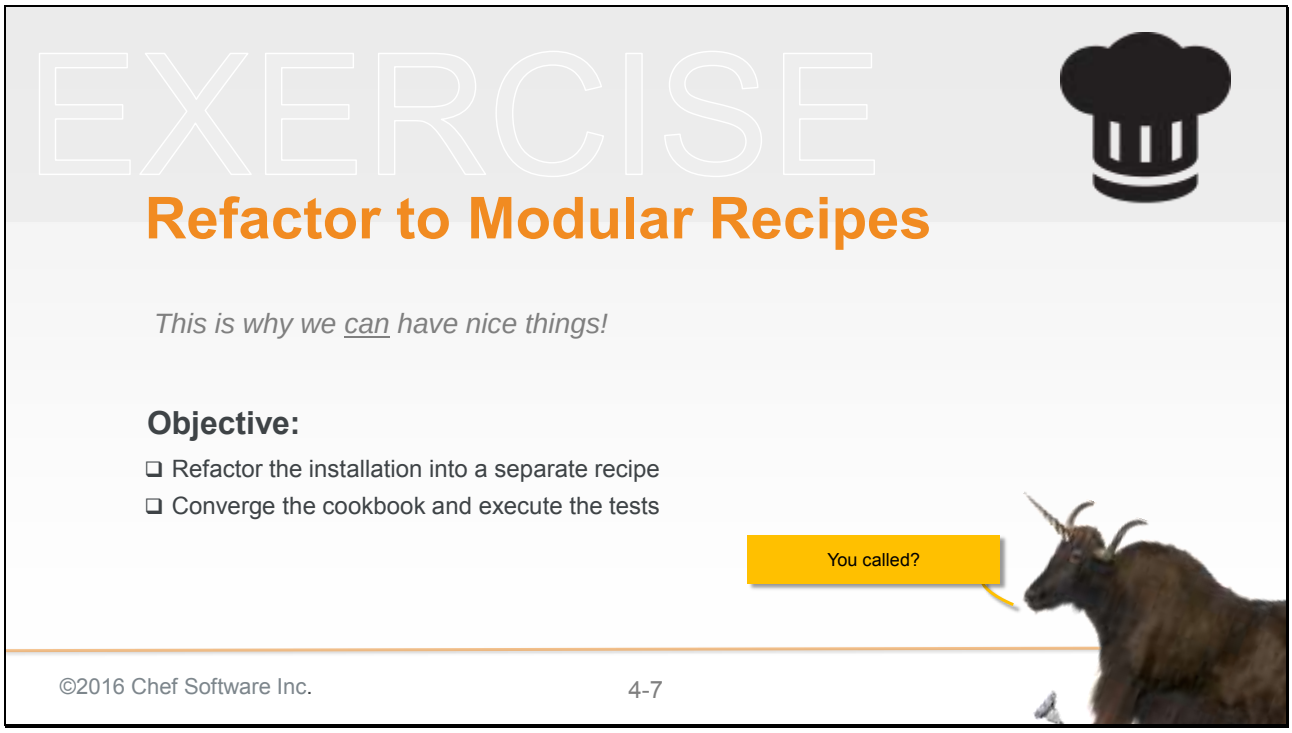
The 'include_recipe' method can be used to include recipes from the same cookbook or external cookbooks. It allows us to accomplish what we saw previously. This gives us the ability to build recipes in more modular ways promoting better re-use patterns within the cookbooks we write.

Slide 6



To allow better re-use we can choose to refactor a single recipe into more modular recipes that focus on their individual concerns. Then these recipes can be included into the original single recipe through the 'include_recipe' method.

Slide 7



This more modular approach to recipes is very common as the complexity of the cookbook continues to grow. The complexity of the cookbook we are developing is not there, nor will it ever be there for the entirety of this course. However, we are still going to use this opportunity to prematurely optimize to demonstrate the refactoring of a cookbook.

Together we will work through creating a recipe that manages the installation of the webserver.

Ask Chef About Generating a Recipe

```
> chef generate recipe --help
```

```
Usage: chef generate recipe [path/to/cookbook] NAME [options]
    -C, --copyright COPYRIGHT        Name of the copyright hol...
    -m, --email EMAIL                Email address of the auth...
    -a, --generator-arg KEY=VALUE    Use to set arbitrary     ...
    -I, --license LICENSE            all_rights, apache2, mit,...
    -g GENERATOR_COOKBOOK_PATH,      Use GENERATOR_COOKBOOK_PA...
        --generator-cookbook
```


CHEF

First let's return to the chef generator tool and it what information it needs to generate a recipe within a cookbook. The recipe generator can be run from within a cookbook or outside of it. If you are within a cookbook you do not need to specify a path to the cookbook; it's optional.

Slide 9

Generate an Install Recipe

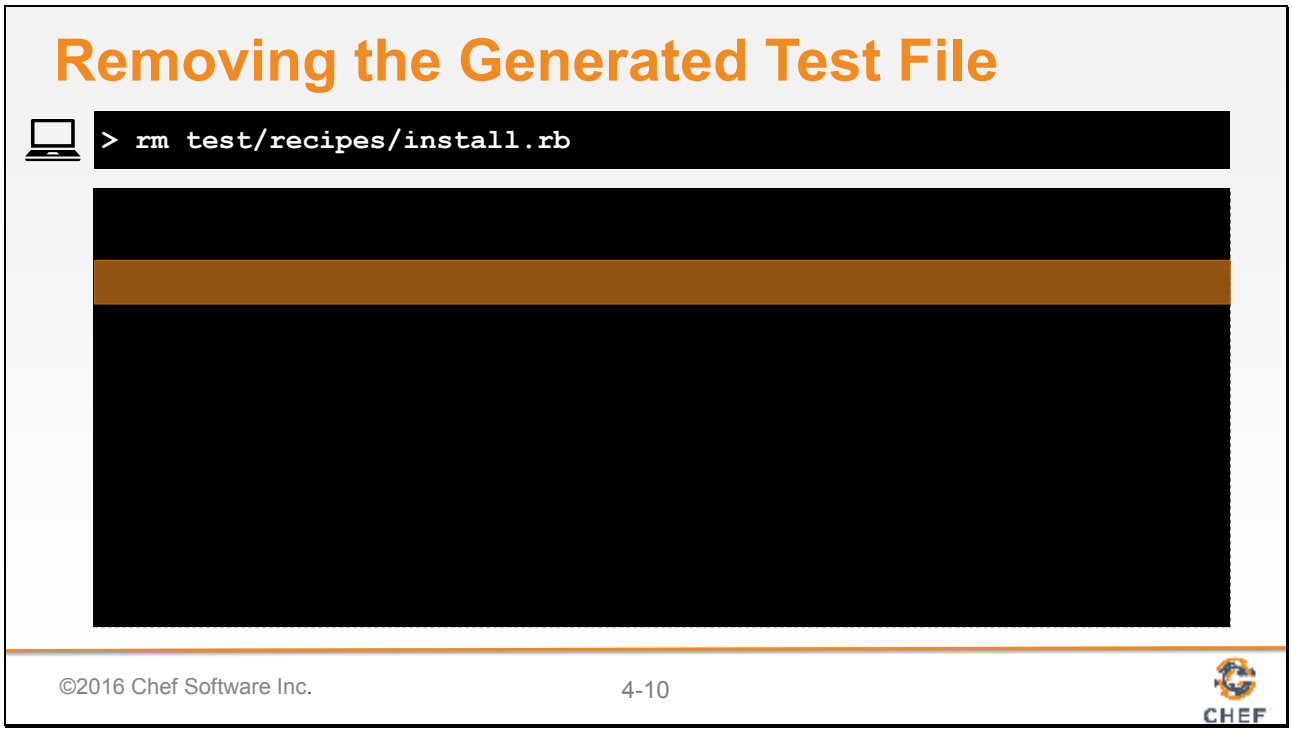
```
> chef generate recipe install
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/httpd/spec/unit/recipes] action create
(up to date)
  * cookbook_file[/home/chef/httpd/spec/spec_helper.rb] action
create_if_missing (up to date)
  * template[/home/chef/httpd/spec/unit/recipes/install_spec.rb]
action create_if_missing
    - create new file
/home/chef/httpd/spec/unit/recipes/install_spec.rb
    - update content in file
/home/chef/httpd/spec/unit/recipes/install_spec.rb from none to
187413
```


Since we are within the cookbook directory you simply need to provide it the name of the recipe you want created.

Instructor Note: The generator will create the recipe file with the recipes directory and also a spec file within the unit test directory. Unit testing is a topic that we will discuss in the next module.

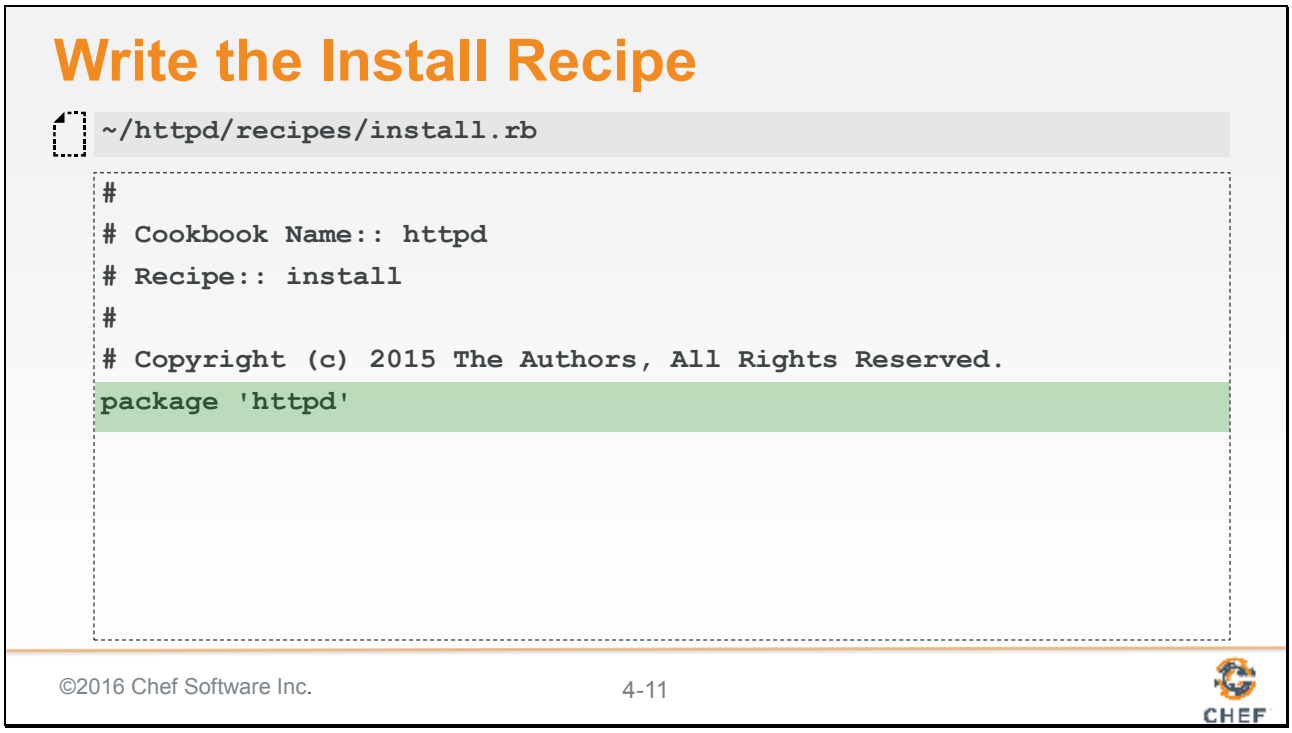
Slide 10



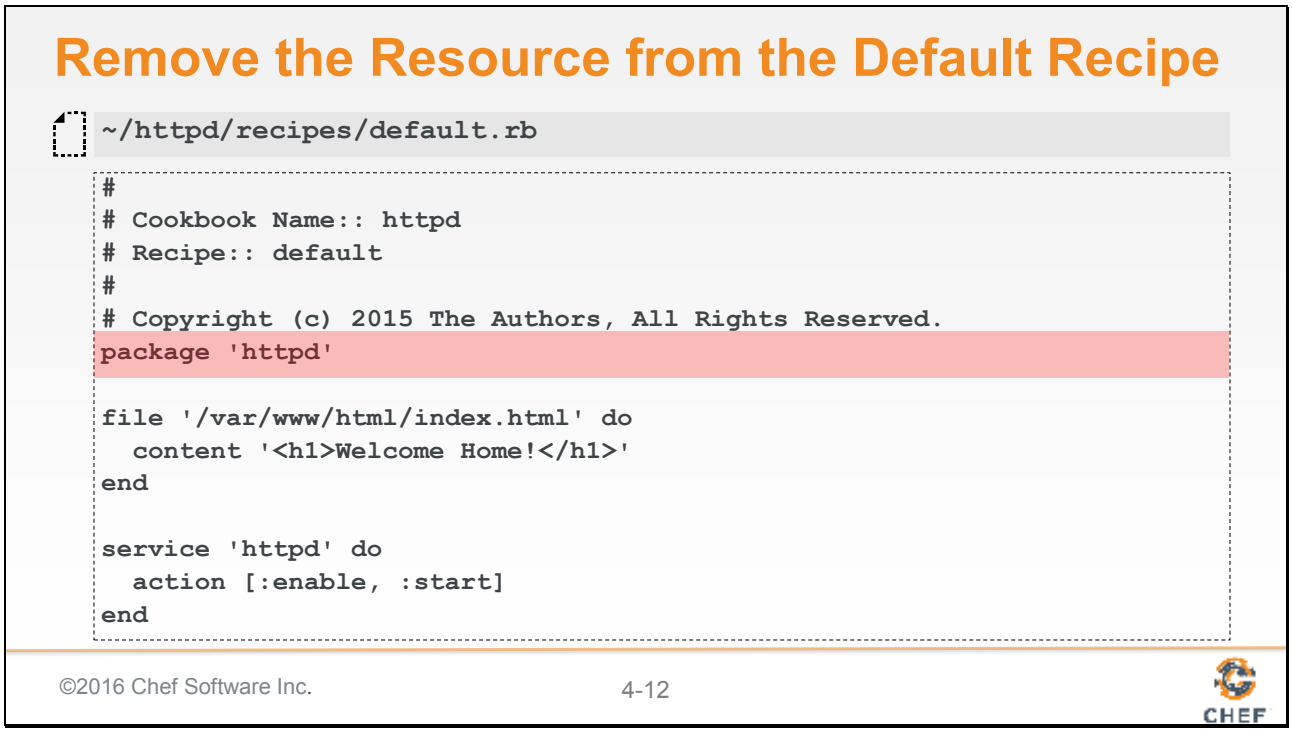
When you use chef, the command-line tool, to generate a recipe it will create three files. First is the recipe file found in the recipes directory. Second is the unit test file found in the 'spec/unit/recipes' directory. Third is the integration test file found in 'test/recipes'.

The test file automatically generate for us contains those same examples we saw in the 'default_test.rb'. We do not want to verify the root user is present and we definitely do not want to verify that port 80 is not listening. So we want to remove this file.

Slide 11

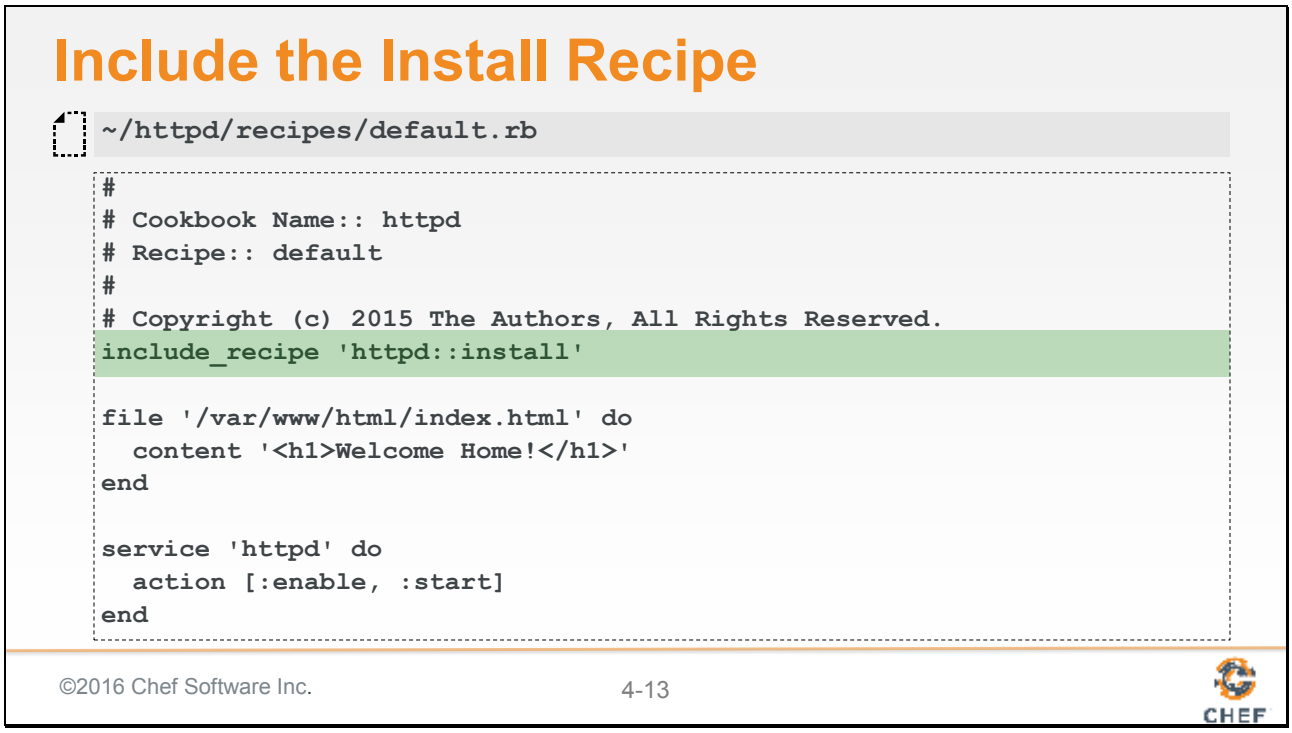


The installation of the web server can be expressed with this one resource. Within the new recipe add the following resource.



Now that we have defined the installation of the webserver in a separate recipe it is time to remove the installation from the default recipe.

Slide 13



Replacing it with the 'include_recipe' method that retrieves the contents of that recipe and includes it here.

Slide 14

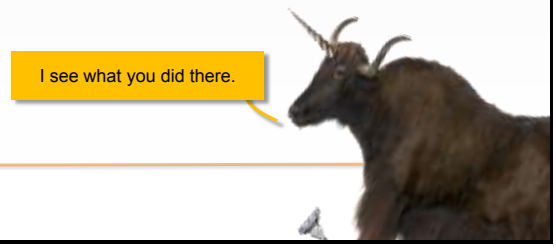
EXERCISE

Refactor to Modular Recipes

This is why we can have nice things!

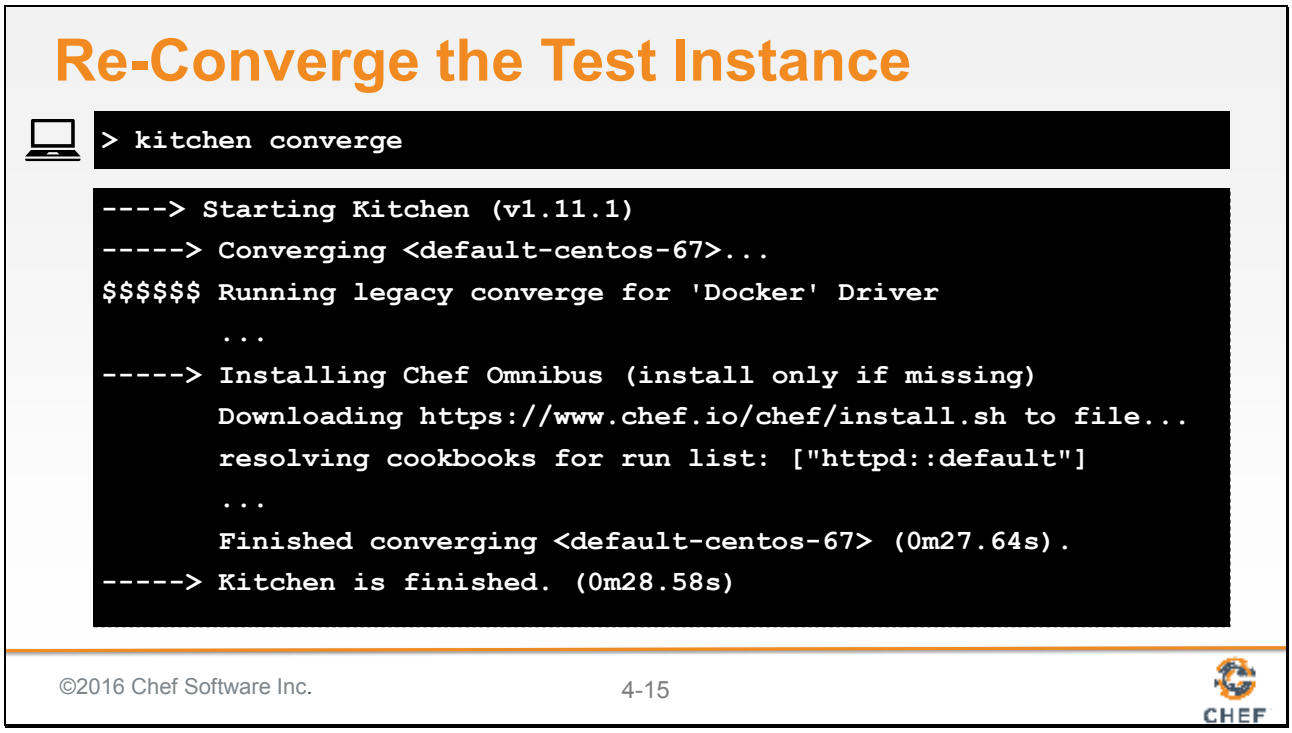
Objective:

- ✓ Refactor the installation into a separate recipe
- ❑ Converge the cookbook and execute the tests



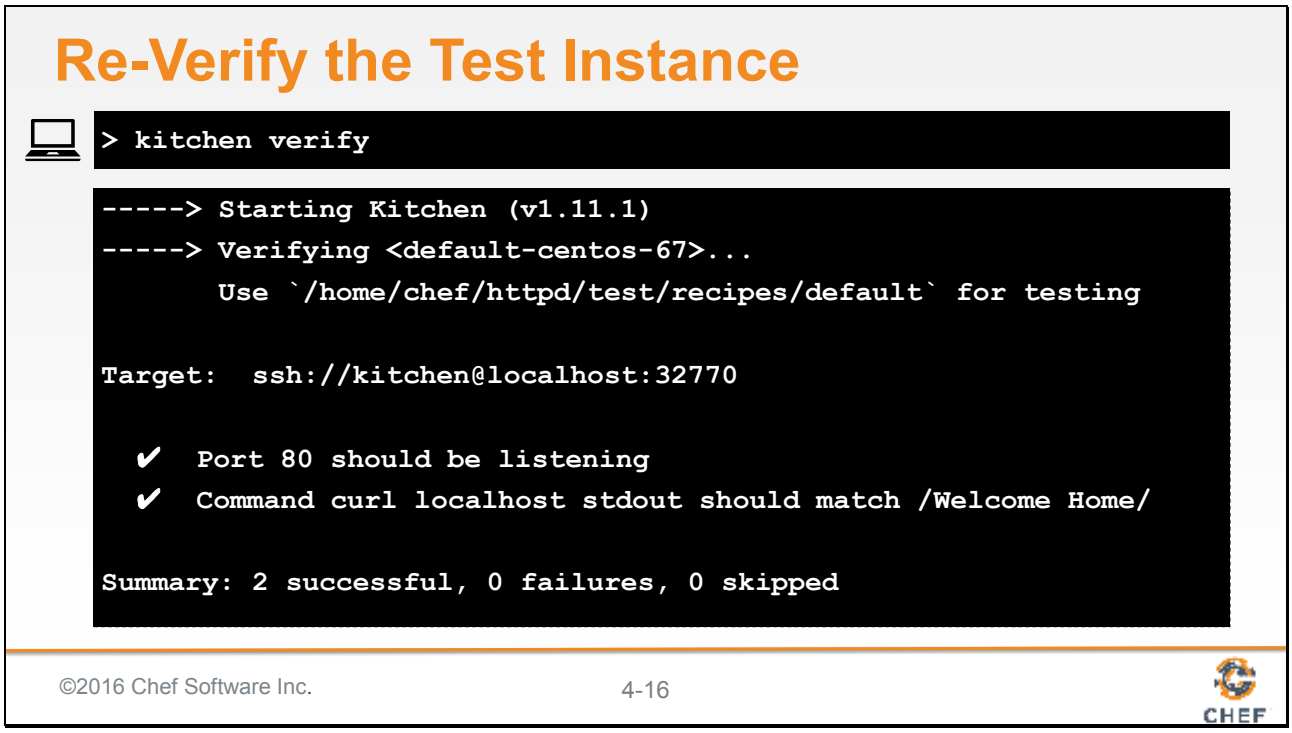
The default recipe has changed. It is now time to ensure that we did everything right by converging the latest changes against the test instance and then verifying the changes by executing our tests.

Slide 15



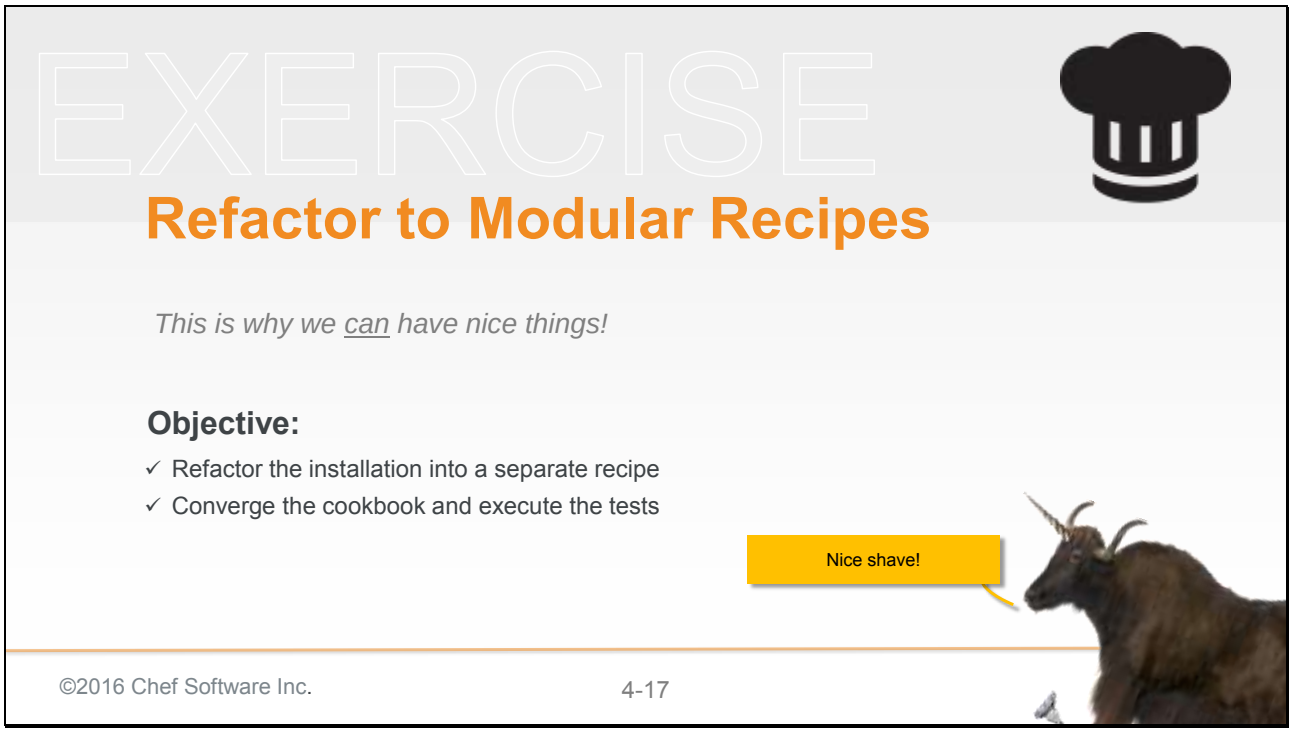
Whenever a change is made to a recipe or component of the cookbook it is important to converge the latest cookbook against the test instance.

If an error occurs that likely means that you have a typo within your default recipe or the install recipe.



If everything converges successfully it is time to verify the state of the instance with the test that we have defined.

Slide 17



Together we were able to refactor the cookbook while implementing the installation recipe.

Slide 18

LAB

The Configuration

- Create a configuration recipe that defines the policy:

```
The file named '/var/www/html/index.html' contains the
content '<h1>Welcome Home!</h1>'.
```

- Remove the file resource from the default recipe
- Converge and verify the test instance to ensure there are no failures

CHEF

Now it is your turn to do the same thing for the webserver configuration. The only configuration that we currently perform for the webserver is write out a new default home page. We still want to move that resource to a separate recipe and ensure that we made the change correctly.

When you are done we will review the next few slides together to review your work.

Instructor Note: Another exercise follows this one to manage the service.

Instructor Note: Allow 5 minutes to complete this exercise.

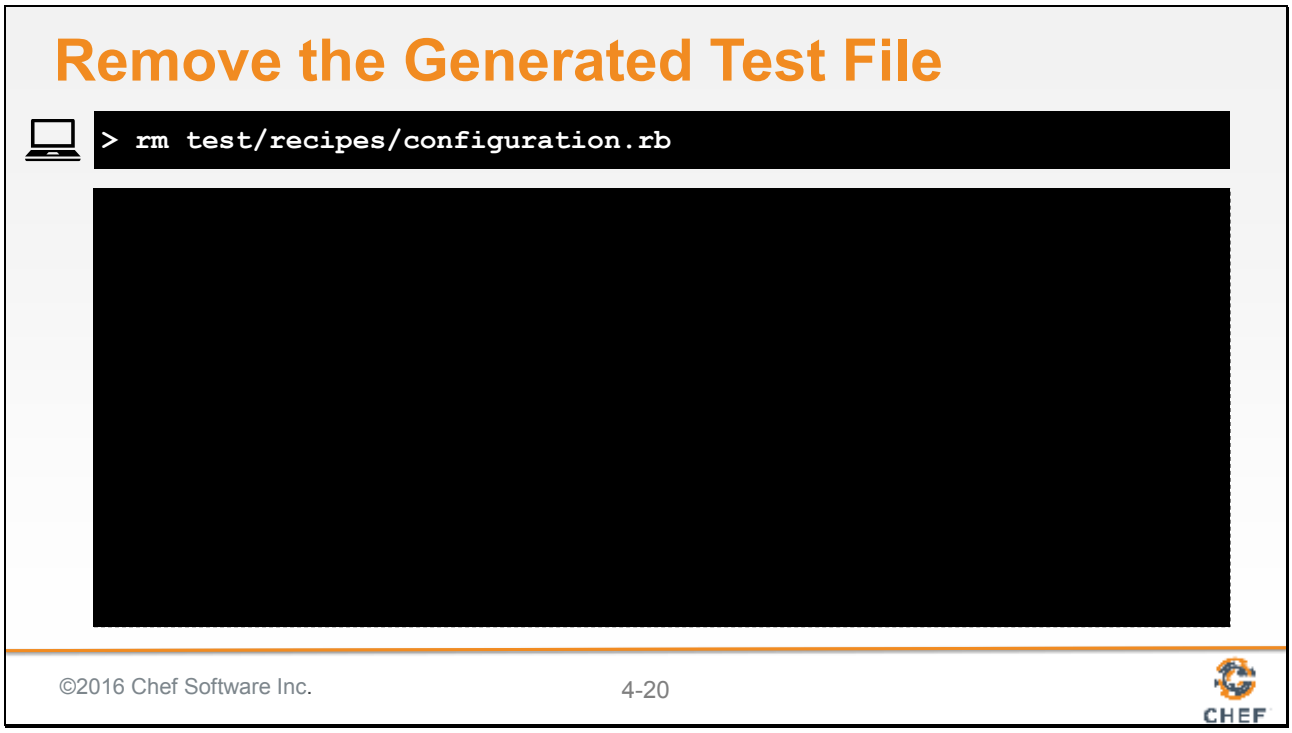
Generate a Service Recipe

```
> chef generate recipe configuration
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/httpd/spec/unit/recipes] action create
(up to date)
  * cookbook_file[/home/chef/httpd/spec/spec_helper.rb] action
create_if_missing (up to date)
  *
template[/home/chef/httpd/spec/unit/recipes/configuration_spec.rb]
action create_if_missing
    - create new file
/home/chef/httpd/spec/unit/recipes/configuration_spec.rb
    - update content in file
/home/chef/httpd/spec/unit/recipes/configuration_spec.rb from none
```


Generate the configuration recipe within the webserver cookbook.

Slide 20



Remove the automatically generated test file as we are not interested in the sample tests that it generates for us.

Slide 21

Write the Configuration Recipe

~/httpd/recipes/configuration.rb

```
#
# Cookbook Name:: httpd
# Recipe:: configuration
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
file '/var/www/html/index.html' do
  content '<h1>Welcome Home!</h1>'
end
```


Define all the resources that are related to the configuration of the webserver within this new recipe

Slide 22

Remove the Resource from the Default Recipe

~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
include_recipe 'httpd::install'

file '/var/www/html/index.html' do
  content '<h1>Welcome Home!</h1>'
end

service 'httpd' do
  action [:enable, :start]
end
```


Remove the resources, that are now defined in the configuration recipe, from the default recipe

Slide 23

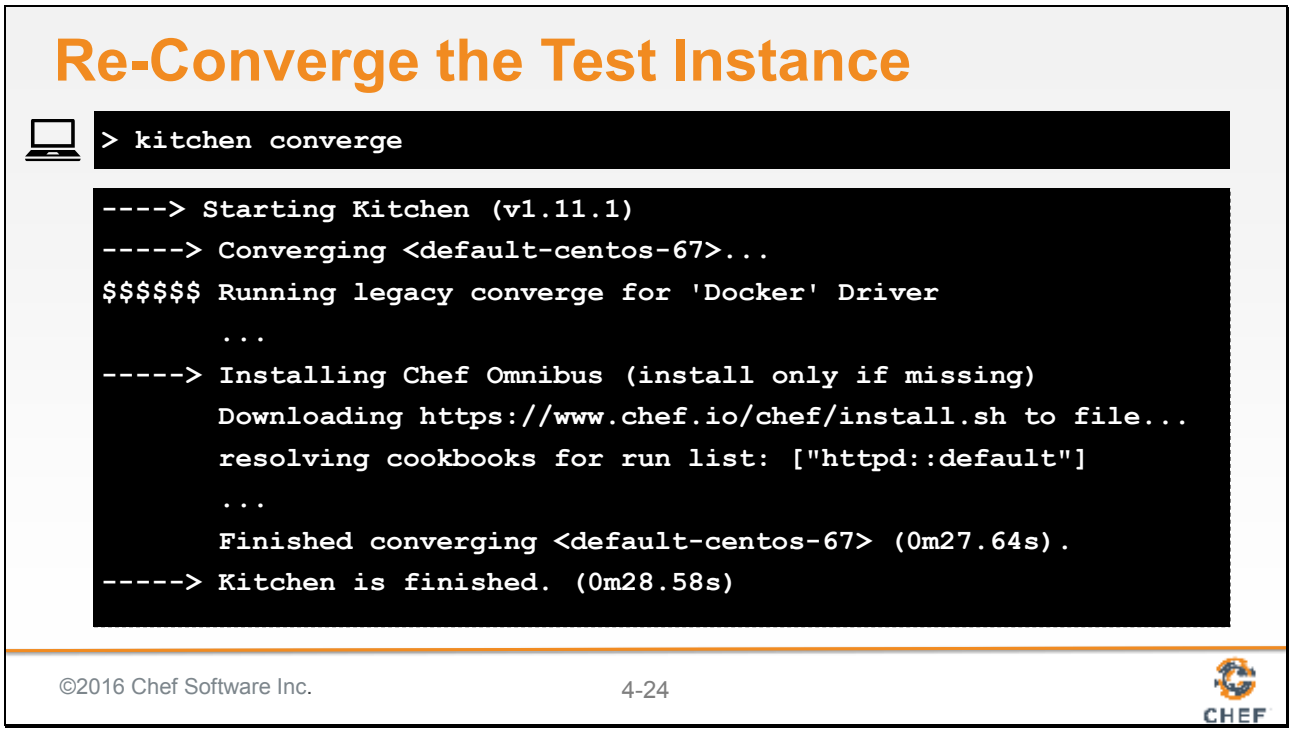
Include the Configuration Recipe

~/httpd/recipes/default.rb

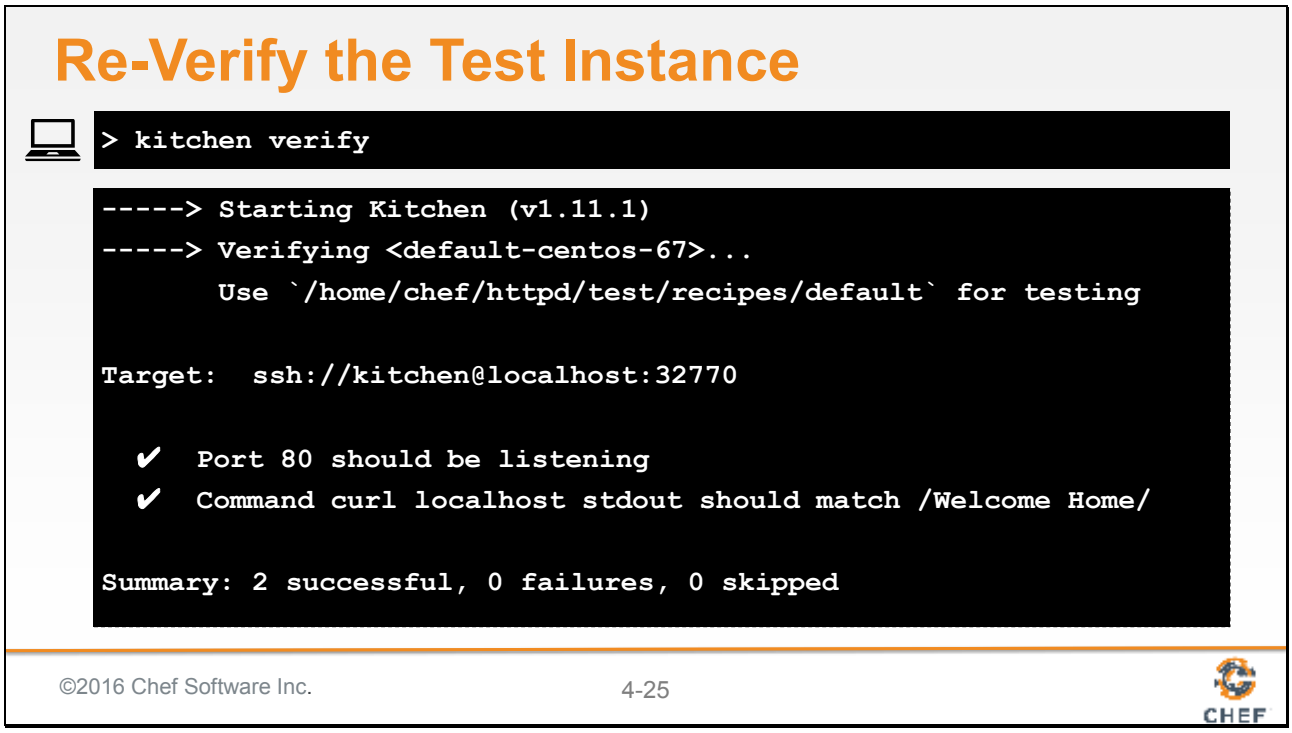
```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
include_recipe 'httpd::install'
include_recipe 'httpd::configuration'

service 'httpd' do
  action [:enable, :start]
end
```


Replace the resources that you have removed with an 'include_recipe' that brings the newly defined configuration recipe.

Slide 24



The recipe changed so it is important to converge the instance.



If everything converges successfully it is time to verify the state of the instance with the test that we have defined.

Slide 26

LAB

The Configuration

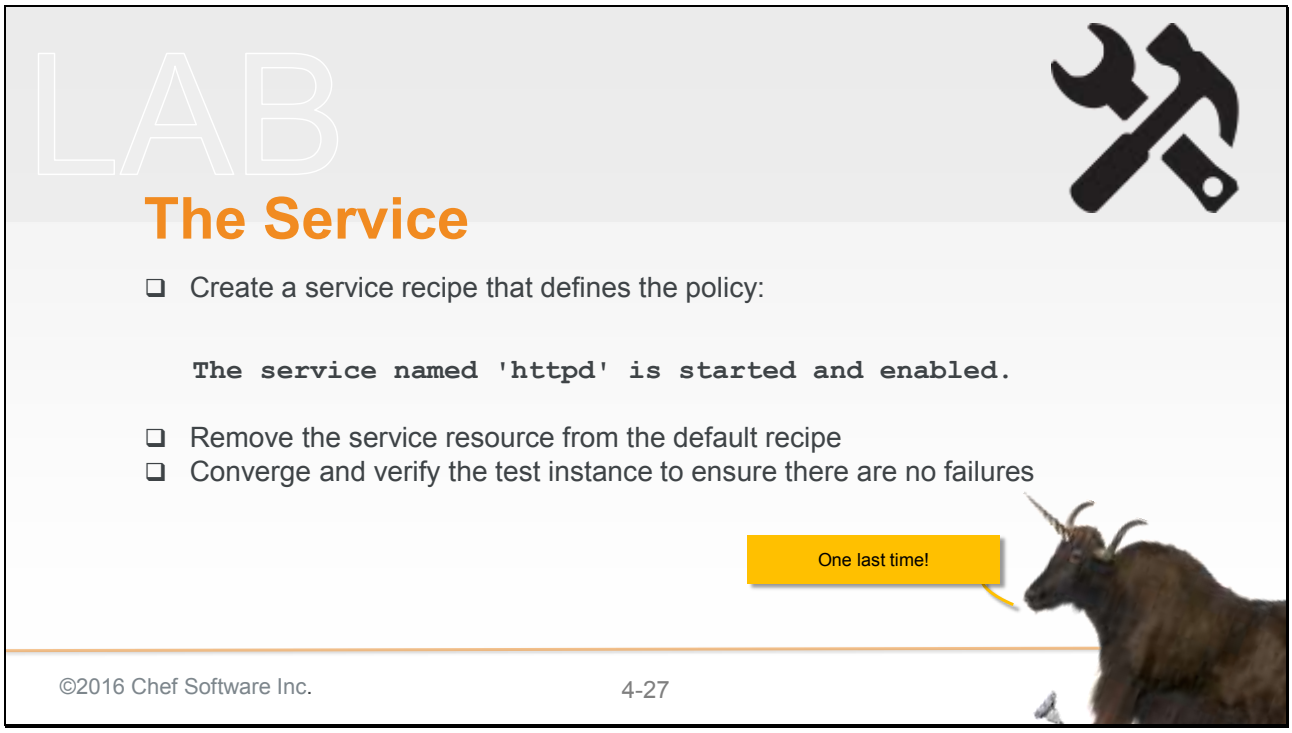
- ✓ Create a configuration recipe that defines the policy:

```
The file named '/var/www/html/index.html' contains the
content '<h1>Welcome Home!</h1>'.
```

- ✓ Remove the file resource from the default recipe
- ✓ Converge and verify the test instance to ensure there are no failures

Congratulations you have successfully refactored the webserver configuration into its own recipe.

Slide 27



Now it is your turn to do the same thing for the webserver service.

When you are done we will review the next few slides together to review your work.

Instructor Note: Allow 5 minutes to complete this exercise.

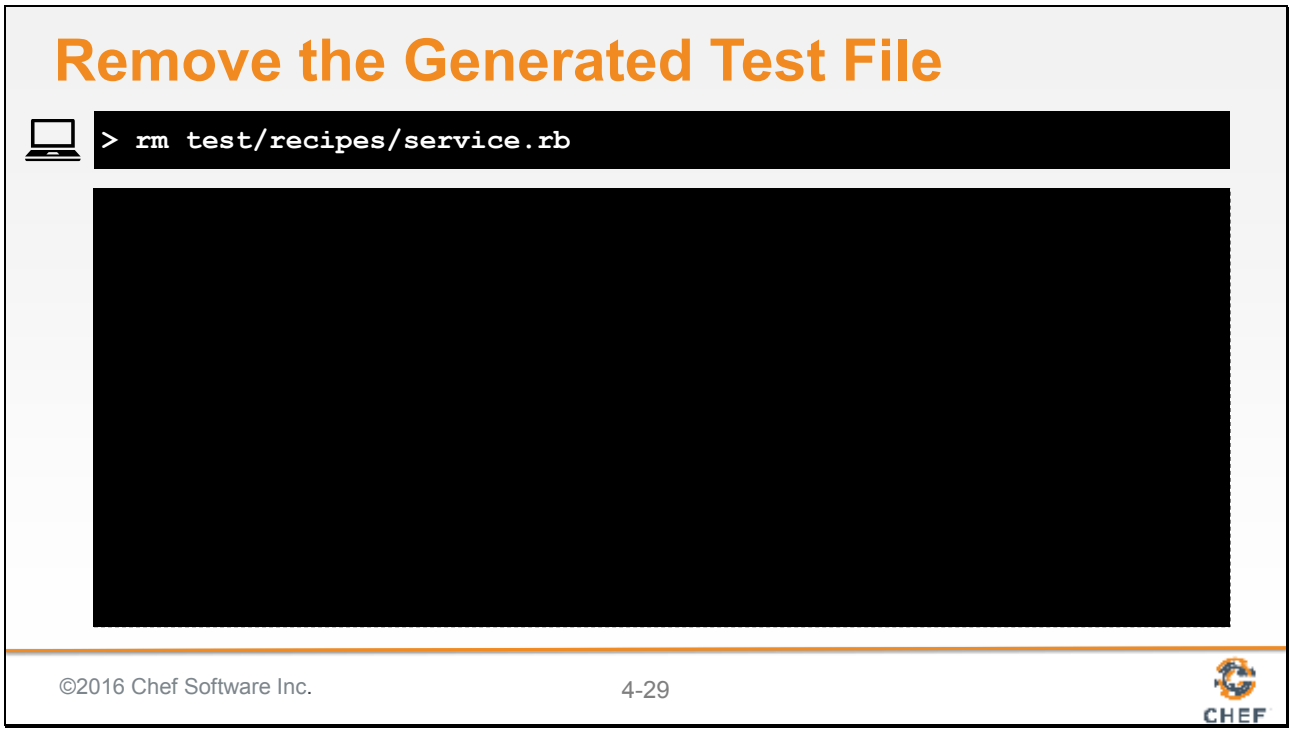
Generate a Service Recipe

```
> chef generate recipe service
```

```
Recipe: code_generator::recipe
  * directory[/home/chef/httpd/spec/unit/recipes] action create
(up to date)
  * cookbook_file[/home/chef/httpd/spec/spec_helper.rb] action
create_if_missing (up to date)
  * template[/home/chef/httpd/spec/unit/recipes/service_spec.rb]
action create_if_missing
    - create new file
/home/chef/httpd/spec/unit/recipes/service_spec.rb
    - update content in file
/home/chef/httpd/spec/unit/recipes/service_spec.rb from none to
1f669c
```


Generate the service recipe within the webserver cookbook.

Slide 29



Remove the automatically generated test file as we are not interested in the sample tests that it generates for us.

Slide 30

Write the Services Recipe

~/httpd/recipes/service.rb

```
#
# Cookbook Name:: httpd
# Recipe:: service
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
service 'httpd' do
  action [:enable, :start]
end
```


Define all the resources that are related to the service of the webserver within this new recipe

Slide 31

Remove the Resource from the Default Recipe

~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
include_recipe 'httpd::install'
include_recipe 'httpd::configuration'

service 'httpd' do
  action [:enable, :start]
end
```


Remove the resources, that are now defined in the service recipe, from the default recipe

Slide 32

Remove the Resource from the Default Recipe

~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
include_recipe 'httpd::install'
include_recipe 'httpd::configuration'
include_recipe 'httpd::service'
```


Replace the resources that you have removed with an 'include_recipe' that brings the newly defined service recipe.

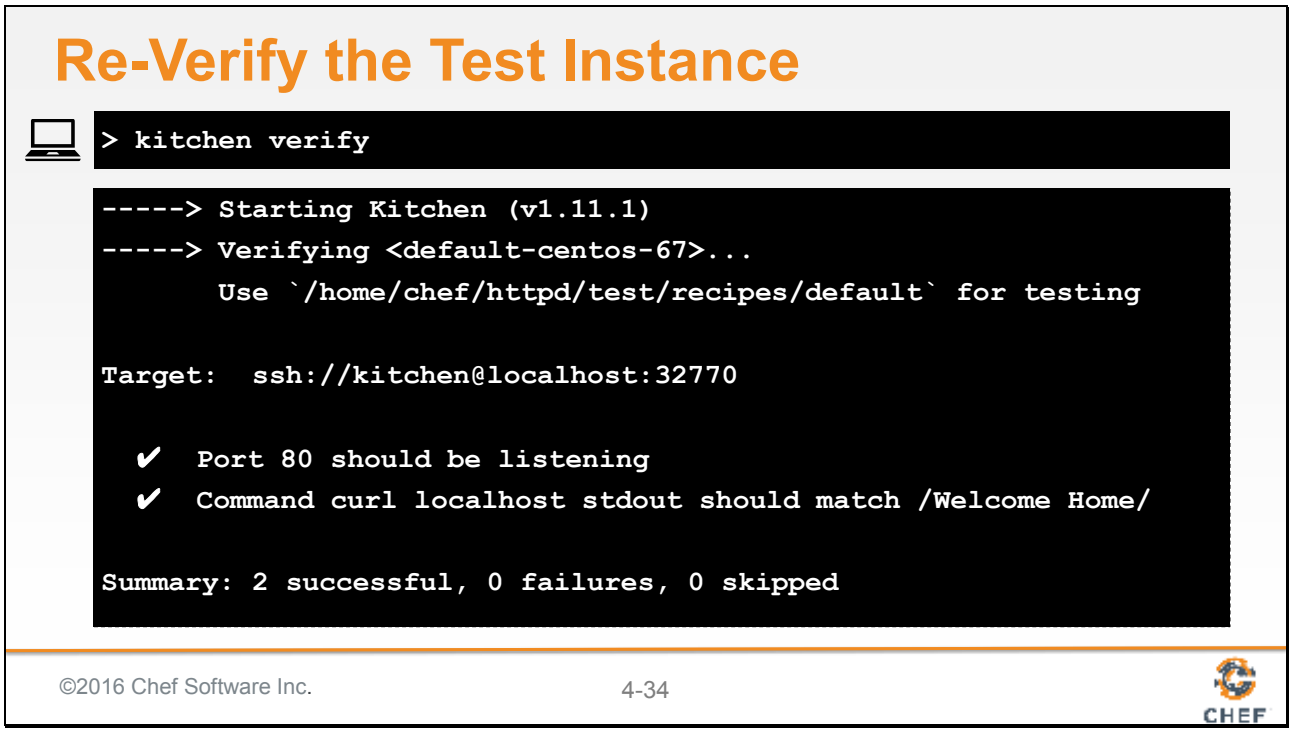
Slide 33

Re-Converge the Test Instance

```
> kitchen converge
```

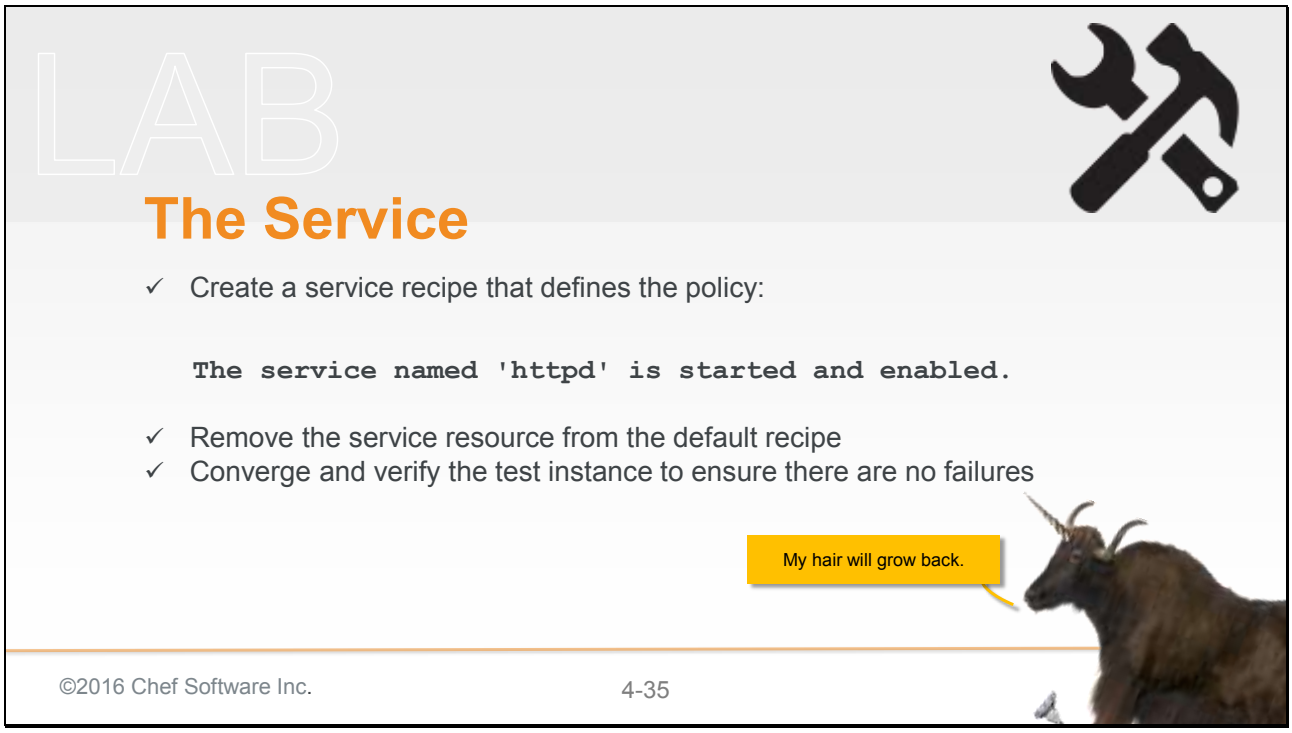
```
----> Starting Kitchen (v1.11.1)
-----> Converging <default-centos-67>...
$$$$$$ Running legacy converge for 'Docker' Driver
       ...
-----> Installing Chef Omnibus (install only if missing)
       Downloading https://www.chef.io/chef/install.sh to file...
       resolving cookbooks for run list: ["httpd::default"]
       ...
       Finished converging <default-centos-67> (0m27.64s).
-----> Kitchen is finished. (0m28.58s)
```


The recipe changed so it is important to converge the instance.



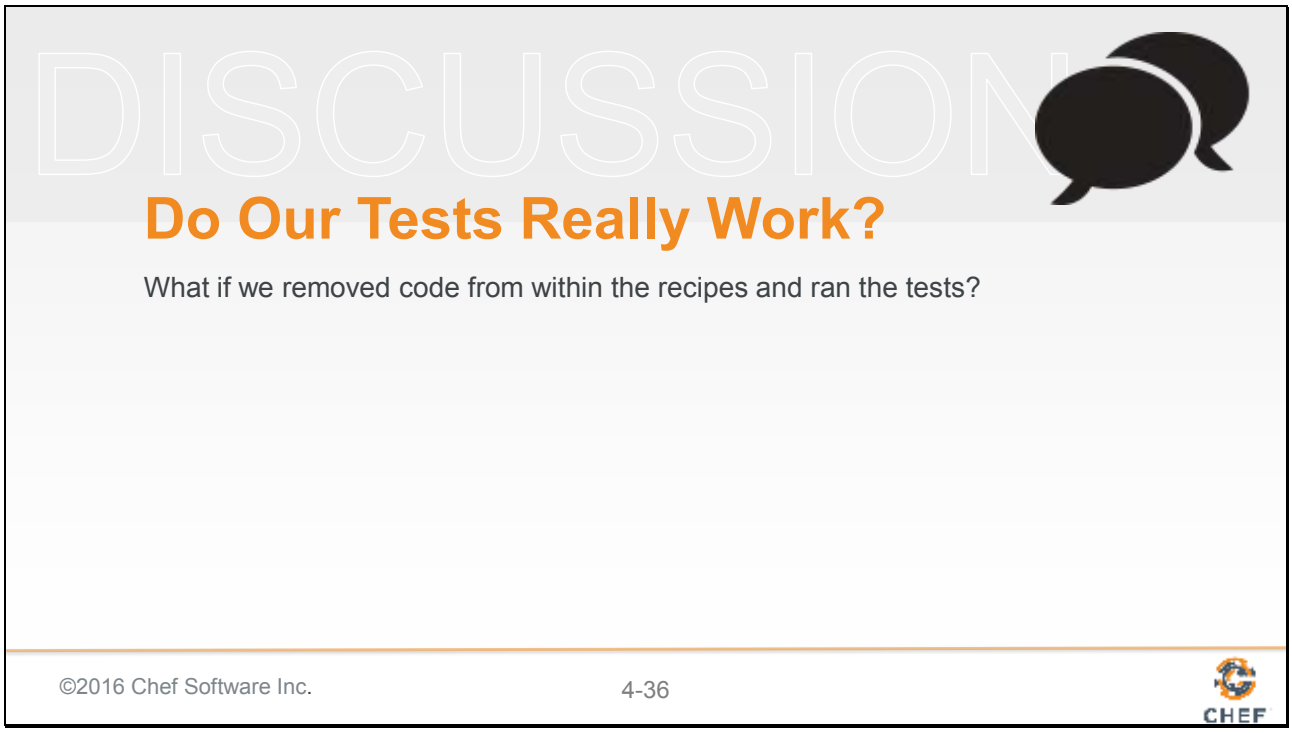
If everything converges successfully it is time to verify the state of the instance with the test that we have defined.

Slide 35



Congratulations you have successfully refactored the webserver service into its own recipe.

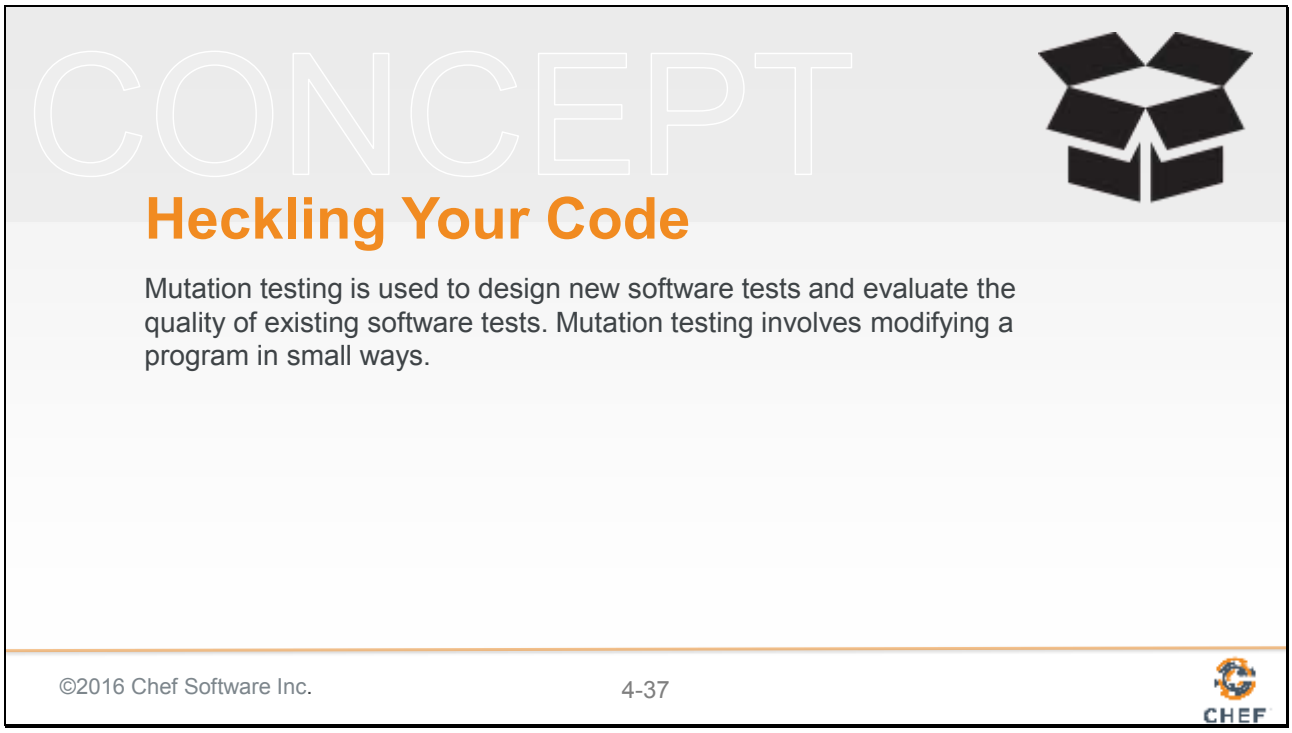
Slide 36



During the group exercise and the lab we made changes to the recipes that we were able to verify on the test instance. If you accidently or purposefully created a typo for yourself you would have seen the converge or the verification fail. However, what if removed code from the recipes that we wrote?

The omission (or in this case removal of code) of resources could have happened. When we refactored the default recipe we may have remembered to remove the resources that manage the configuration but forgot to use the 'include_recipe' to ensure we loaded the new recipe. Or it is possible that we created a service recipe that we never populated but made all the appropriate changes to the default recipe.

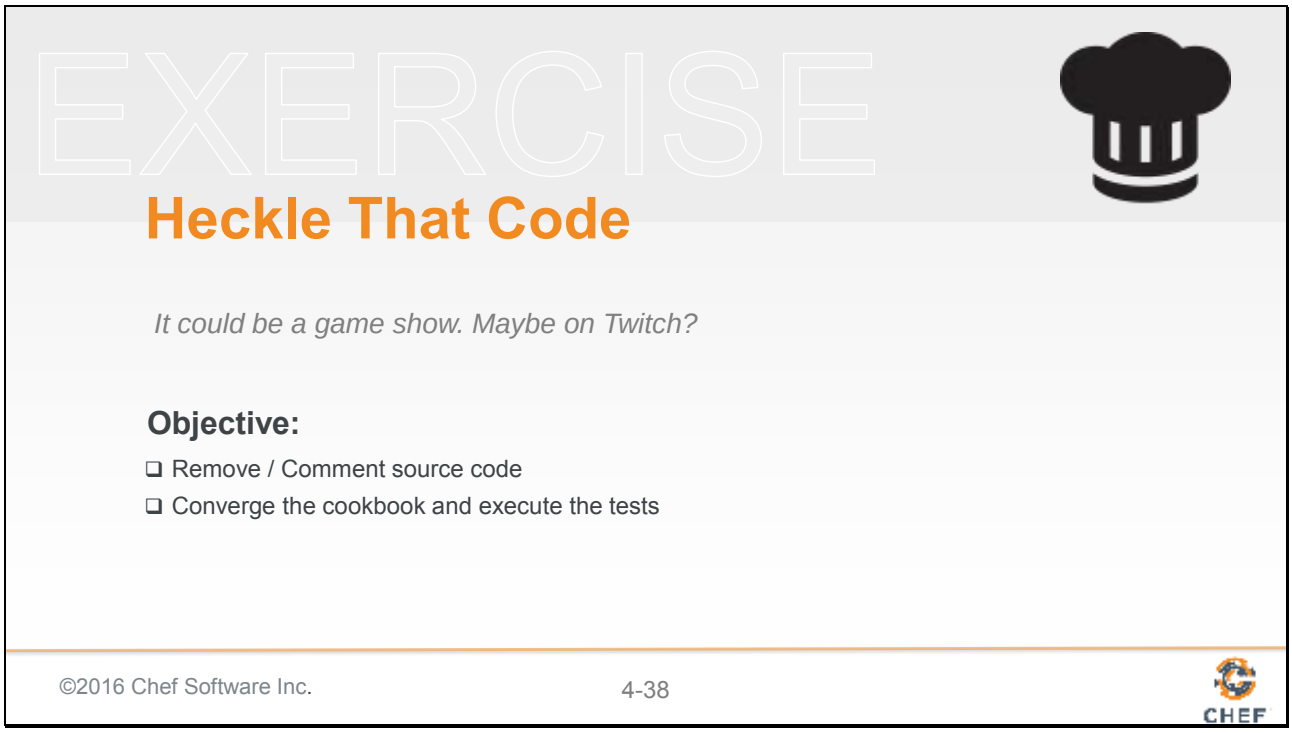
Slide 37



Removing code sabotages the policy that you have defined. If you used Test Kitchen to converge and verify the cookbook and saw a failure you can sleep soundly at night knowing your tools have you covered. On the other hand, if Test Kitchen were to return success, after such a change, then it might cause you to break out in a cold sweat.

Removing code from a recipe or recipes is a small change. So is introducing a typo into the code, specifying a different resource name or changing the value of a resource attribute. The process of modifying the code in small ways and then executing the test suite against it is often times referred to as mutation testing.

Slide 38



Before we leave this module, let's do a little mutation testing, to ensure the test that we have defined is good enough.

Slide 39

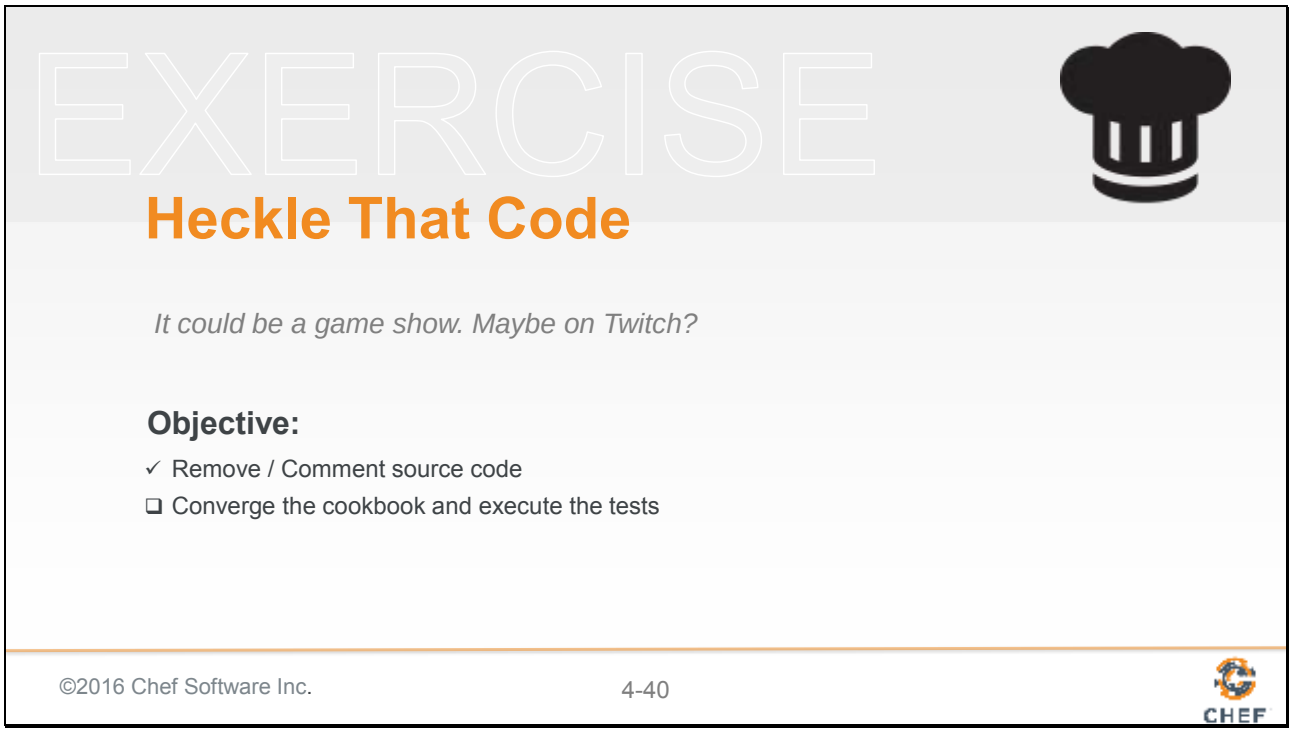
Comment Out Key Code Within the Default Recipe

~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2015 The Authors, All Rights Reserved.
# include_recipe 'httpd::install'
include_recipe 'httpd::configuration
include_recipe 'httpd::service'
```

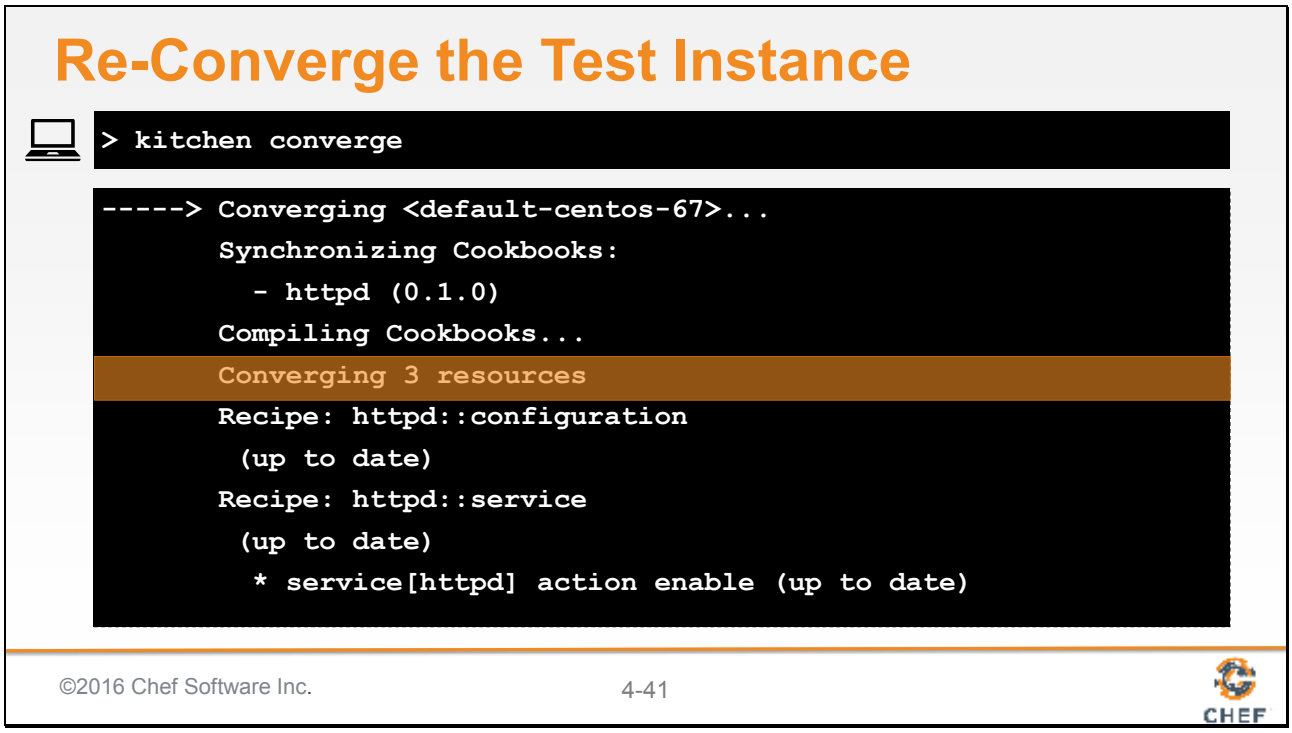

Return to the default recipe and choose one line to remove or comment out. Here I have chosen to comment out the first line that includes the install recipe.

Slide 40

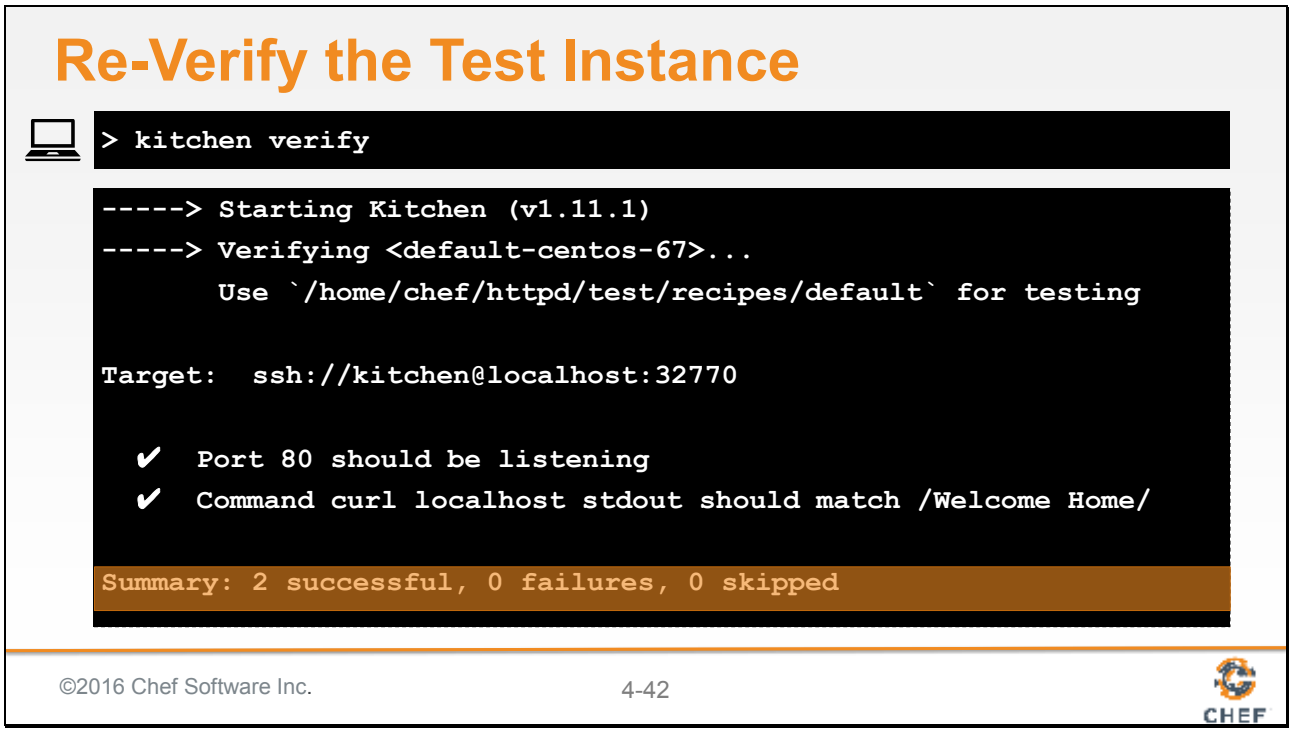


Now with that small mutation in place it is time to converge the cookbook and execute the tests.

Slide 41

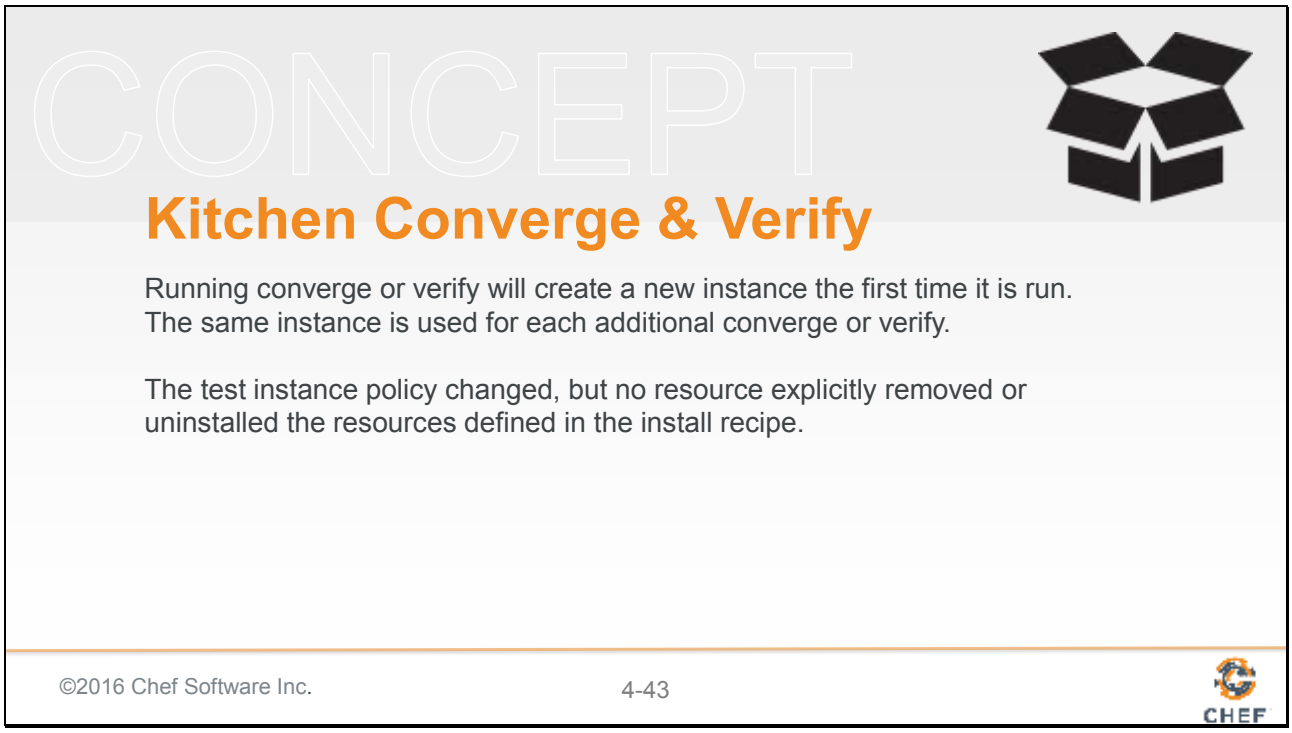


When converging the updated recipe it no longer shows the install recipe being loaded. This has changed the number of resources that are converged on the test instance. Removing the recipe from the default recipe does not remove any of the components that it previously installed.



Verification of the test instance will return a success. Despite removing the install recipe from the default recipe the test instance is still able to serving the default web page that our test is looking for when it requests data from the site.

Slide 43

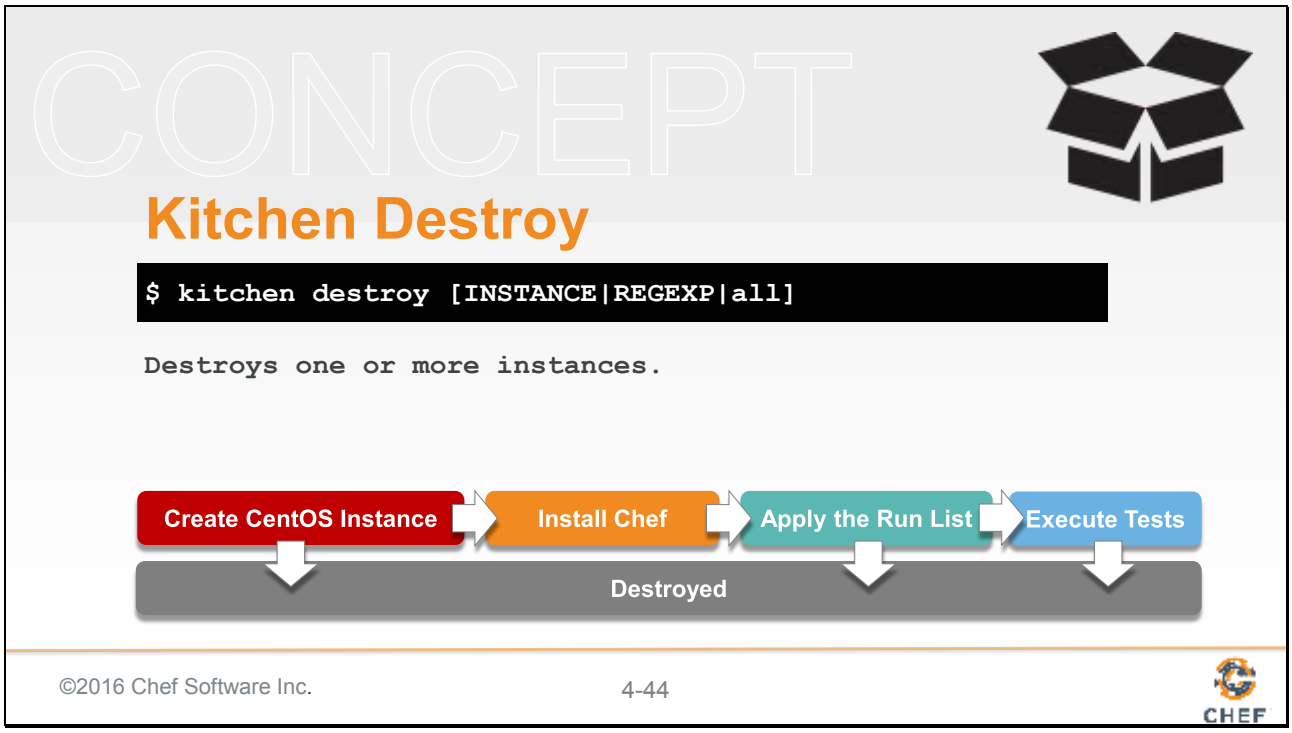


This is important feature and limitation of using Test Kitchen's 'converge' and 'verify'. Both of these commands will create a test instance the first time they are executed. Every time after these commands will use the same test instance again and again.

When we remove resources from a recipe we do not explicitly uninstall them from the test instance. We simply do not enforce their policy anymore. On an existing system, which this test instance is after the first run, this means it is actually in the desired state that we no longer define. That means that the webserver is still installed, the default web page has still been updated, and the service is still running.

To ensure our cookbook works on a new system it is important to delete the test instance and start over.

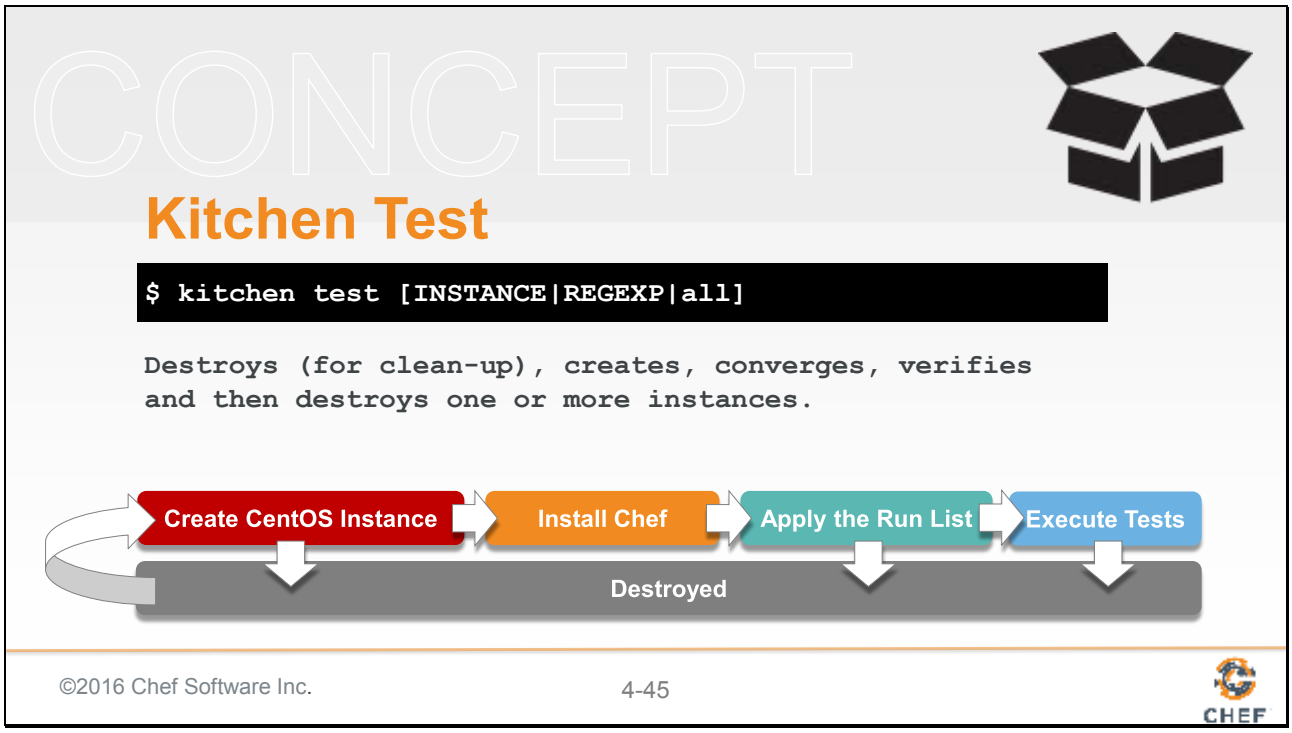
Slide 44



Test Kitchen provides the 'destroy' subcommand. Destroy is available at all stages and essentially cleans up the instance. This is useful when you make changes to the configuration policy you define and you want to ensure that it will work on a brand new instance.

Instructor Note: It works as all the other commands do with regard to parameters and targeting instances.

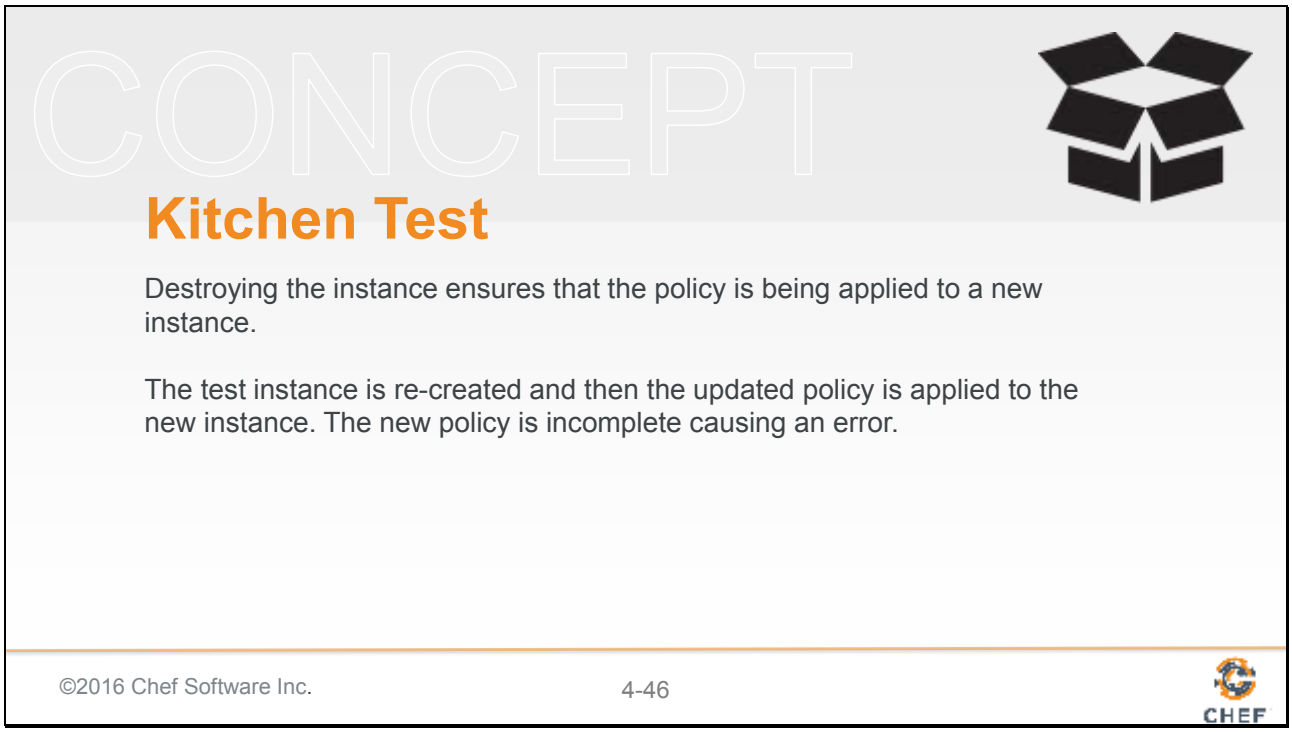
Slide 45



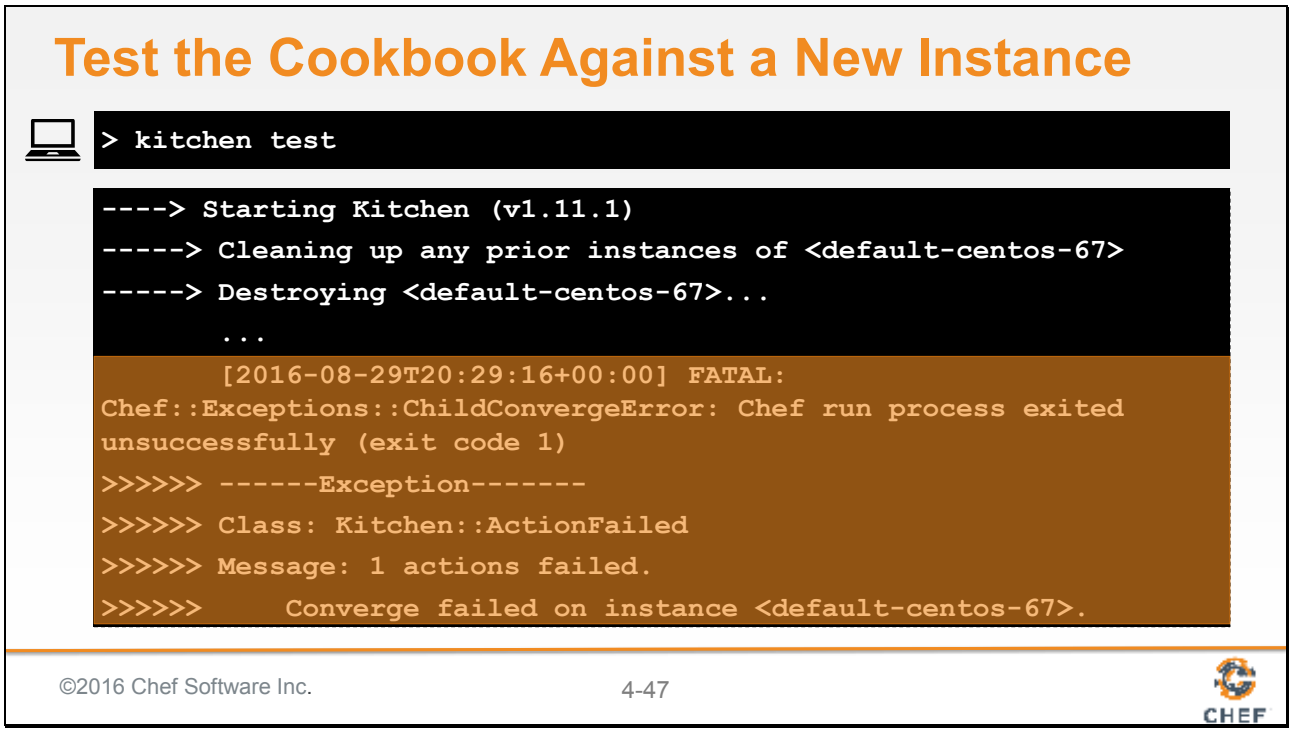
Test Kitchen also provides the subcommand 'test'. Test provides one command that wraps up all the stages in one command. It will destroy any test instance that exists at the start, create a new one, converge the run list on that instance, and the verify it. If everything passes the 'test' subcommand will finish by destroying that instance. If it fails at one of these steps it usually leaves the instance running to allow you to troubleshoot it.

Instructor Note: It works as all the other commands do with regard to parameters and targeting instances.

Slide 46



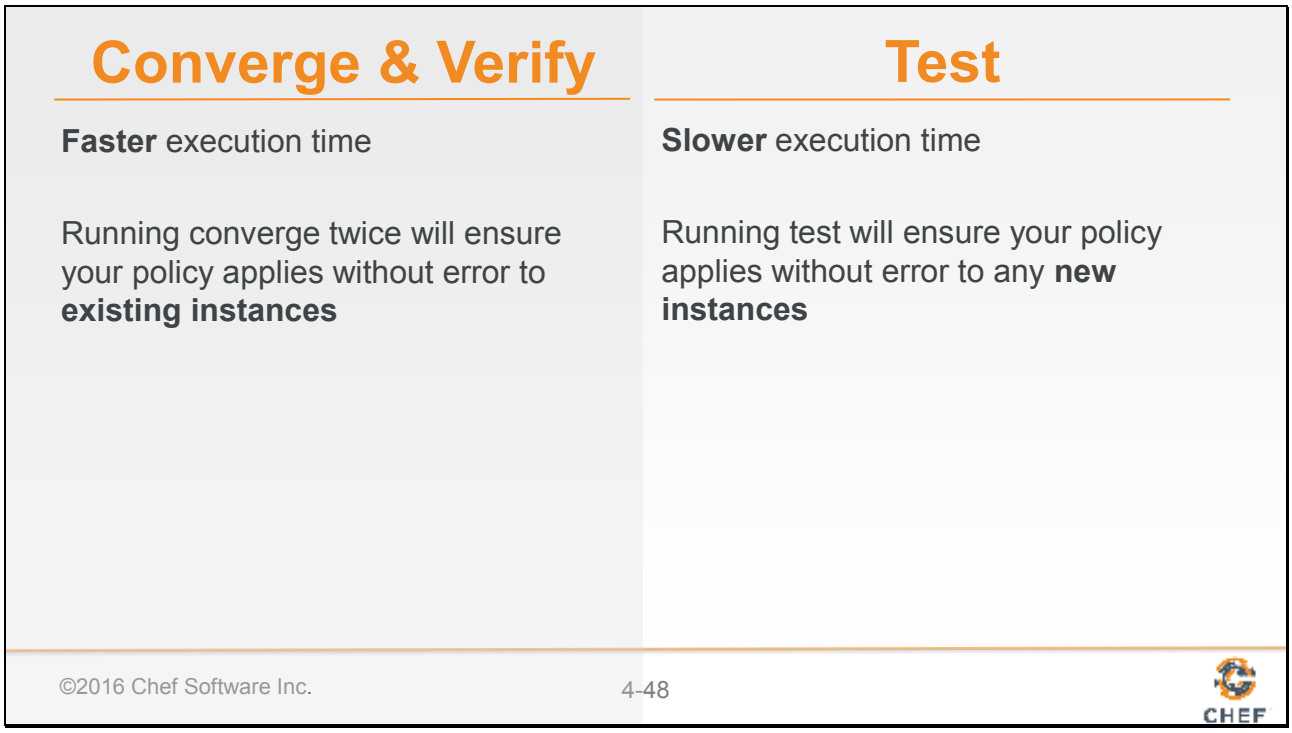
Running 'kitchen test' is useful if want to ensure the policy you defined works on a new instance.



Running 'kitchen test' in this instance will expose the issue that we created by removing that installation of the webserver. This is because the new instance no longer installed the necessary packages so the file path was never created for the default HTML file and there are no services to run.

The test that you wrote correctly verifies the state of the system. What is important to notice is that there are important differences in the Test Kitchen commands.

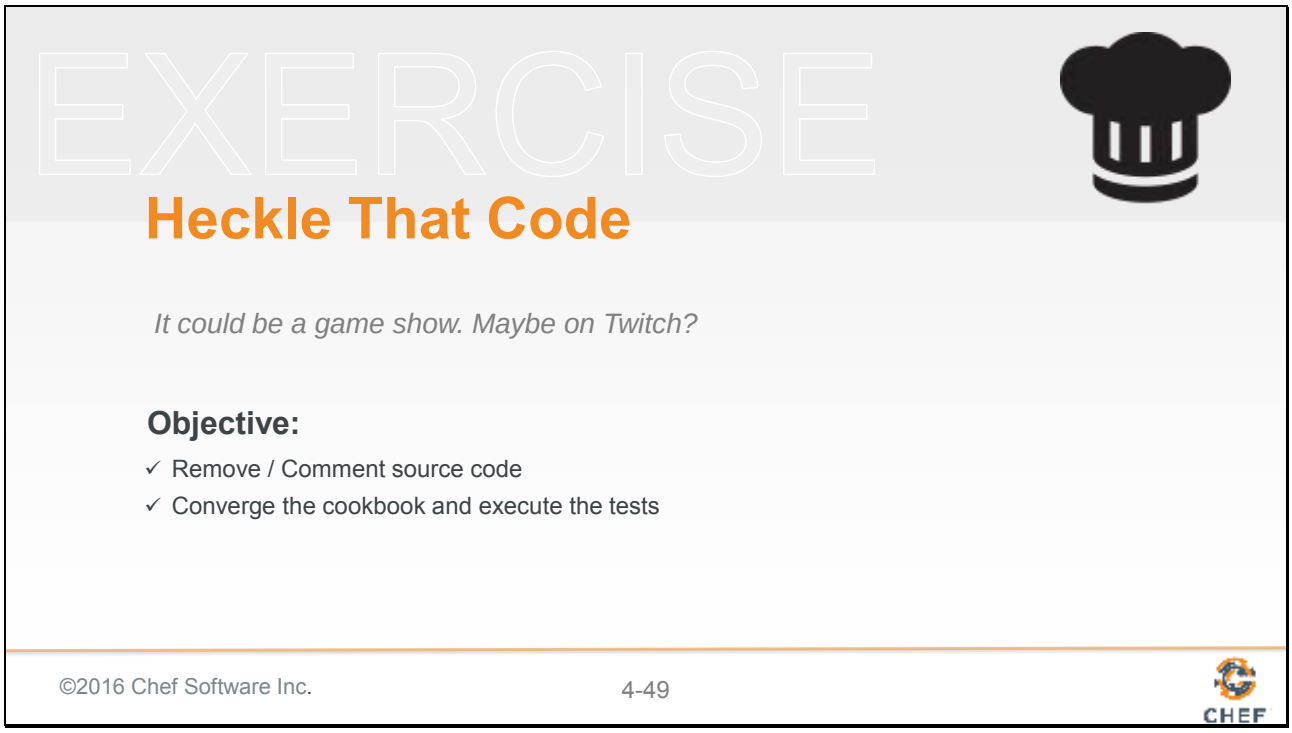
Slide 48



Using Test Kitchen to run 'kitchen converge' and 'kitchen verify' is much faster because you are essentially applying and verifying the policy that you have defined against an already running instance. The drawback is that only running 'converge' and 'verify' will not demonstrate for you how your policy will act on a brand new instance.

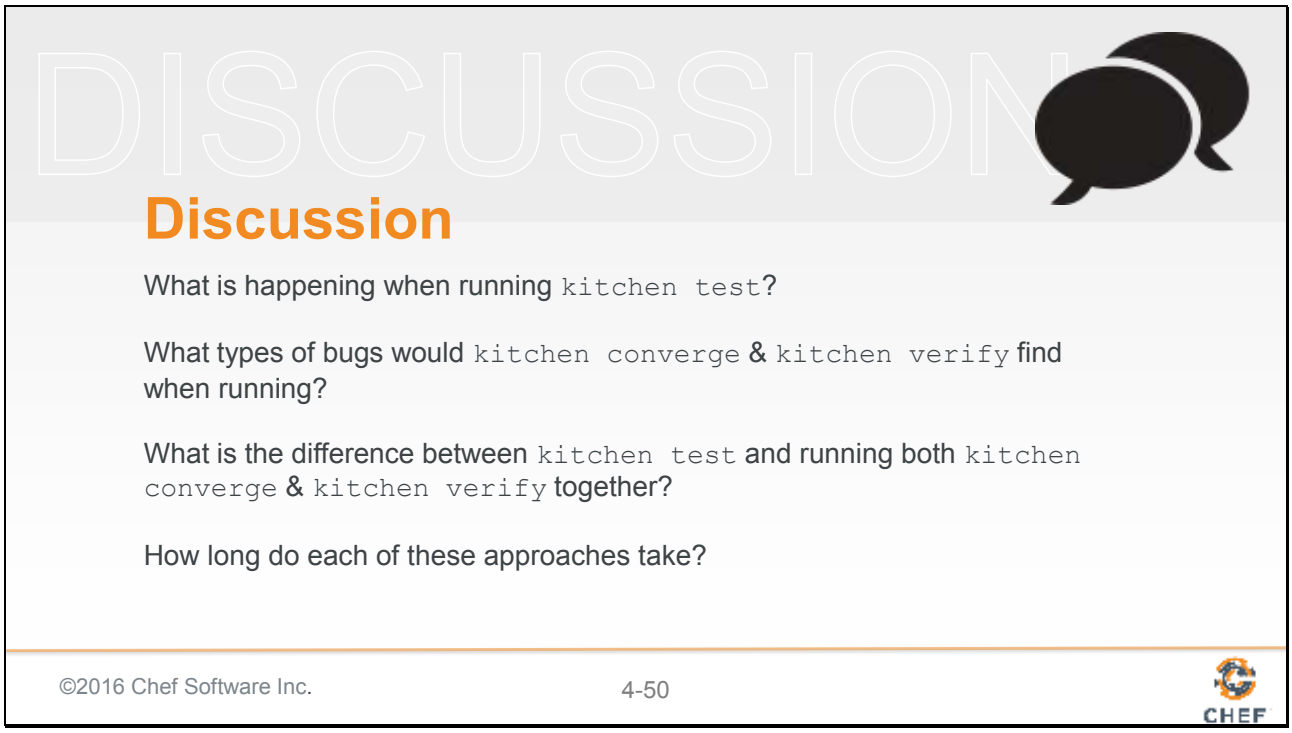
Using Test Kitchen to run 'kitchen test' is slower because every time you are recreating the test instance, installing chef, and applying the policy on that new instance. The drawback here is the longer feedback cycle and only running 'test' will not demonstrate for you how your policy will act on an existing instance.

Slide 49



Removing code and causing a failure showed us some of the differences between 'kitchen converge and verify' and 'kitchen test'. To ensure that we understand these important differences let's have a discussion.

Slide 50

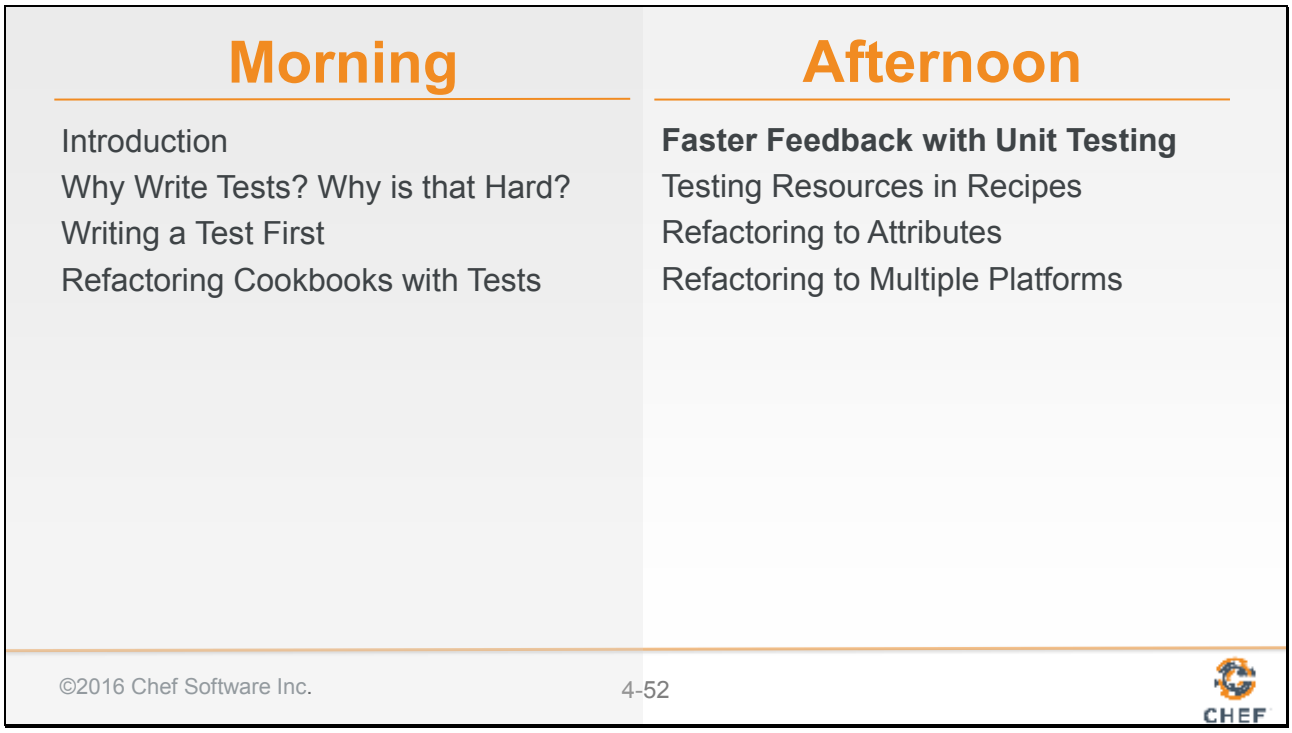


Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 51

Before we complete this section, let us pause for questions.

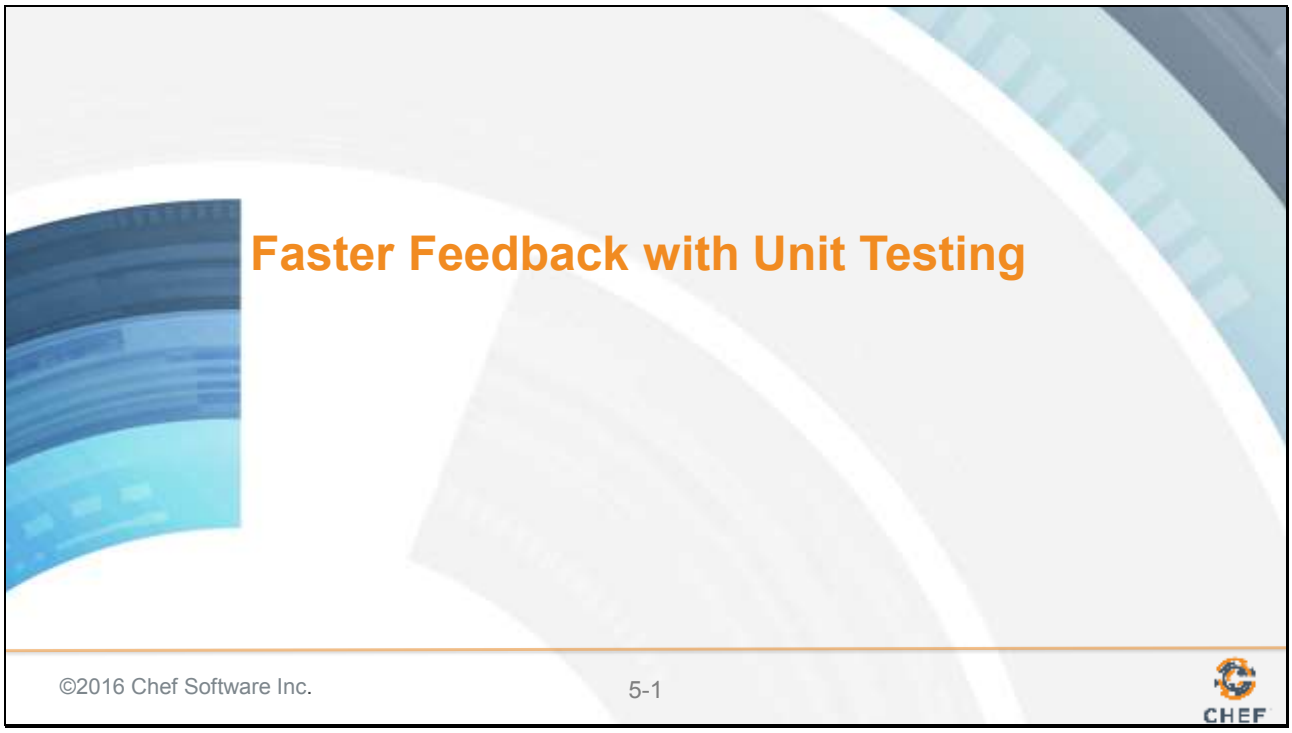
Slide 52



You have performed the complete TDD cycle from start to finish. Now that you have seen this cycle and understand that we are simply going to continue to repeat it as we develop cookbooks it is important to talk about the amount of time it takes for us to get feedback. In the next section we are going to explore that further by introducing a new testing tool and language that promises to give us faster feedback.

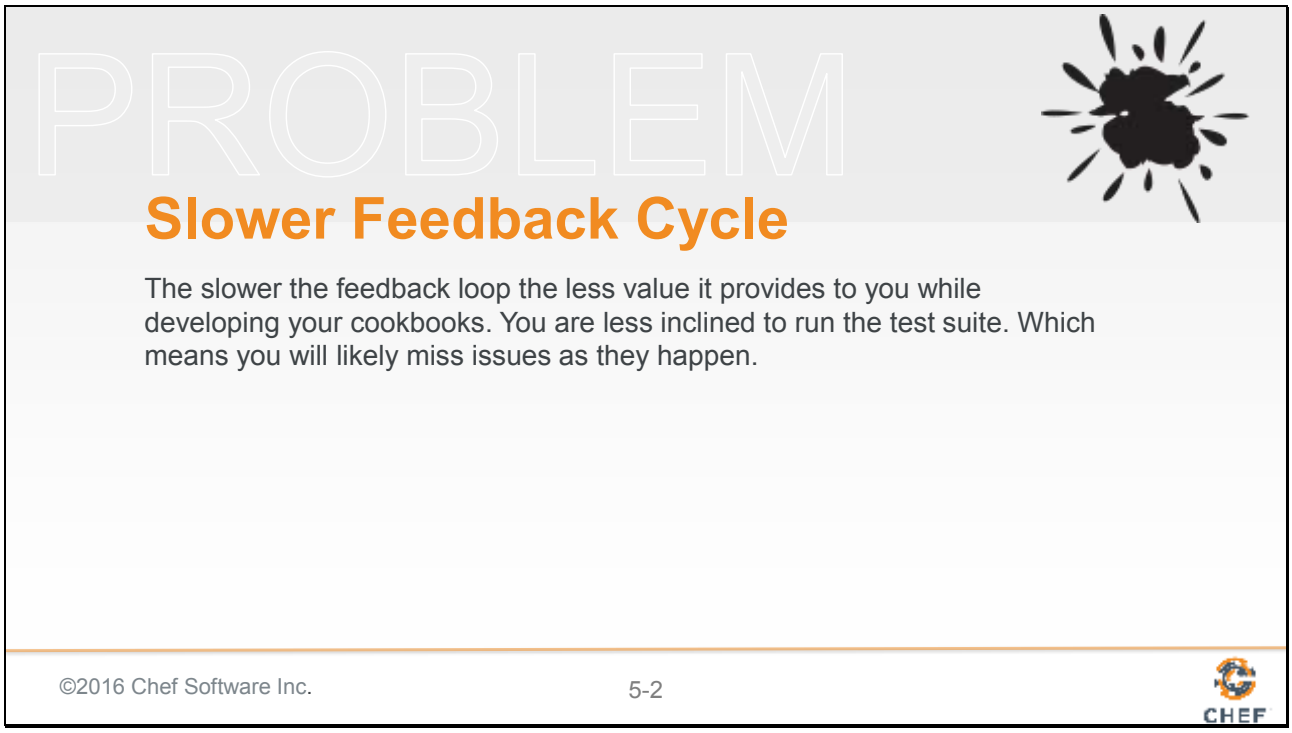
Slide 53

5: Faster Feedback with Unit Testing



If you are planning on adopting Test Driven Development and use it to validate most if not all all of the changes that you make to a cookbook you now have to are welcoming into your workflow the interruption of running the tests. Testing provides value as it validates the work that you accomplish but it is still an interruption.

Slide 2



Interruptions are not conducive to helping you building a flow. To help reduce the interruptive nature of testing we can look at ways to decrease the amount of time you have to wait to receive the feedback from the tests. A faster feedback cycle will increase your likelihood of seeking that feedback again for smaller sets of changes. Slower feedback cycles will increase your likelihood of seeking feedback less often. Causing you create larger sets of changes which has the chance of masking potential issues.

Slide 3

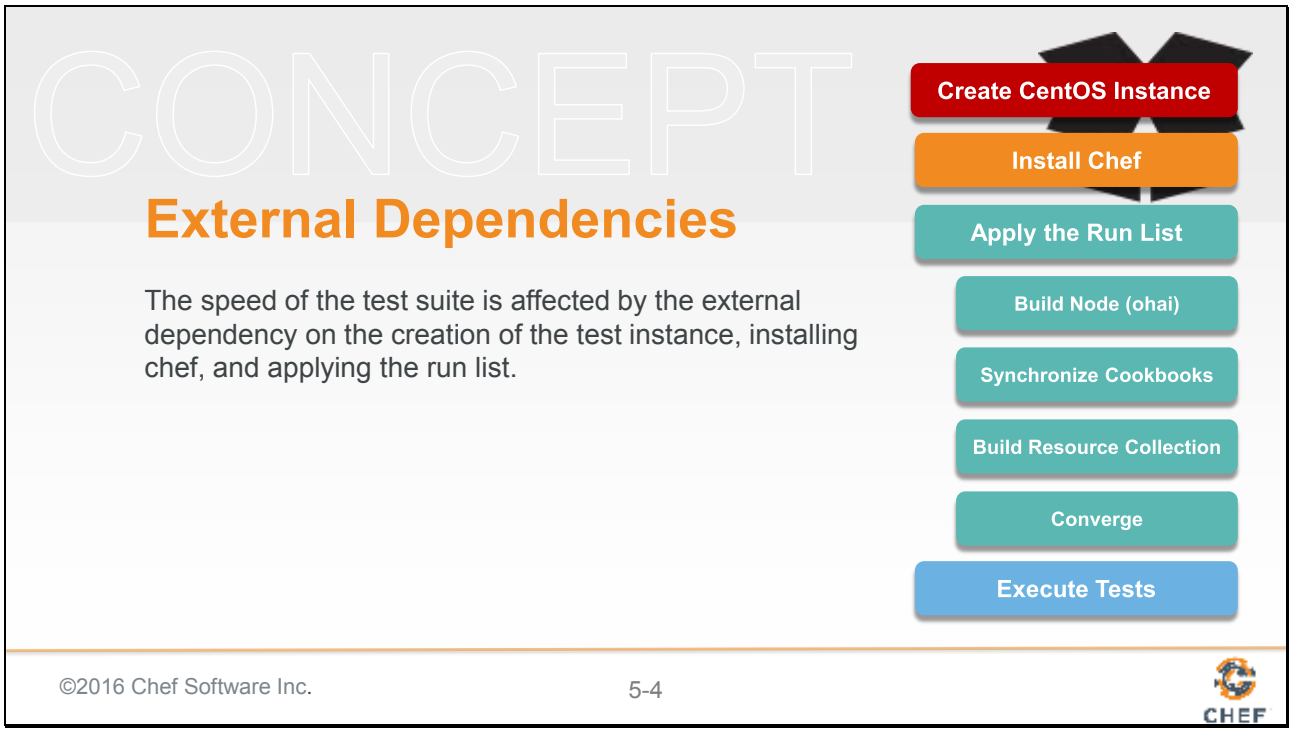
Objectives

After completing this module, you should be able to:

- Explain the importance and limitations of unit testing
- Write and execute a unit test

In this module you will learn the importance and limitations of unit testing as you write and execute unit tests to help increase the rate at which you receive feedback.

Slide 4

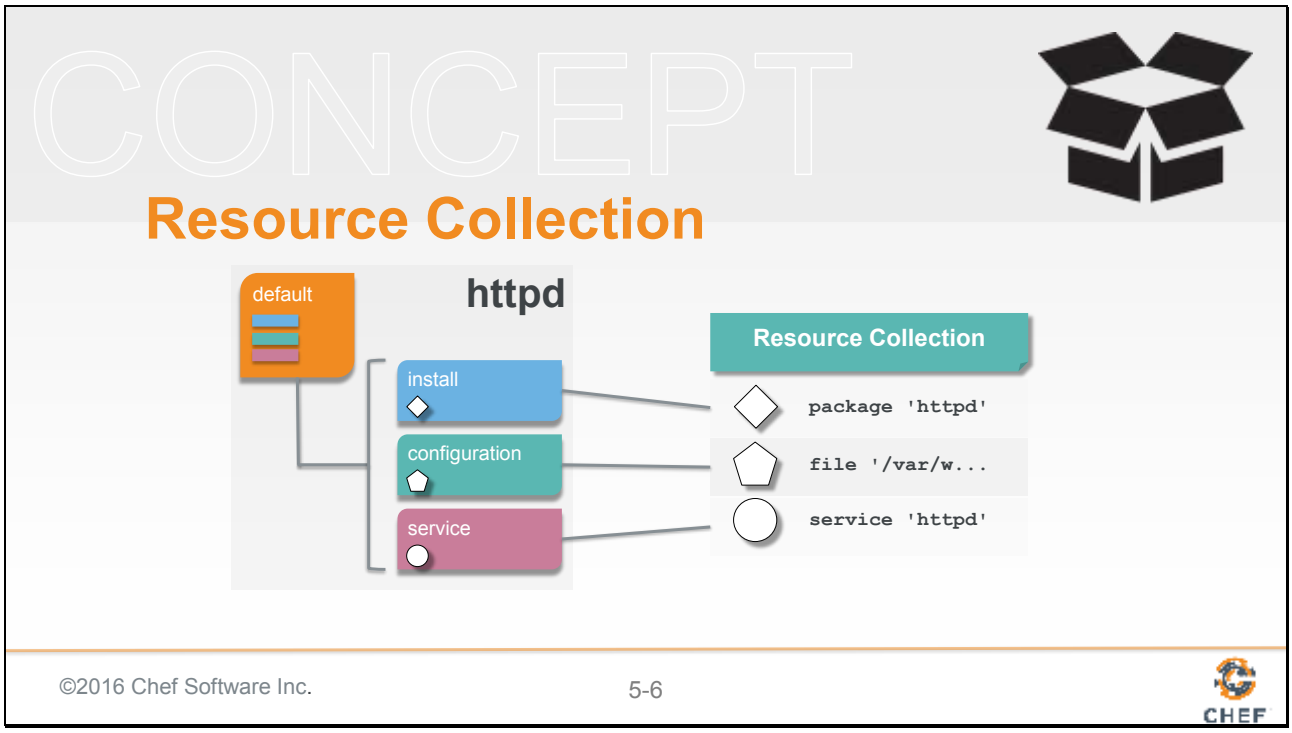


The reason that the feedback cycle takes as long as it does with Test Kitchen is because of the external requirements. Creating the test instance, installing chef, and then applying the run list provide real value because we are able to see the recipe being applied to a virtual instance. However, all these external dependencies incur a time cost as we wait for the network to download images or packages, the test instance's processor to calculate keys or data, or the file-system to create files and folders.

Slide 5

When we mutated our code and executed the test suite we created issues with the resources that we defined and recipes that we included. These changes affected the resources that were applied to the system by omitting resources from the 'Resource Collection'. If we were able to remove the external dependencies and focus on the state of the Resource Collection we would be able to determine if there were problems with the recipes we wrote without the need of any of those external dependencies.

Slide 6

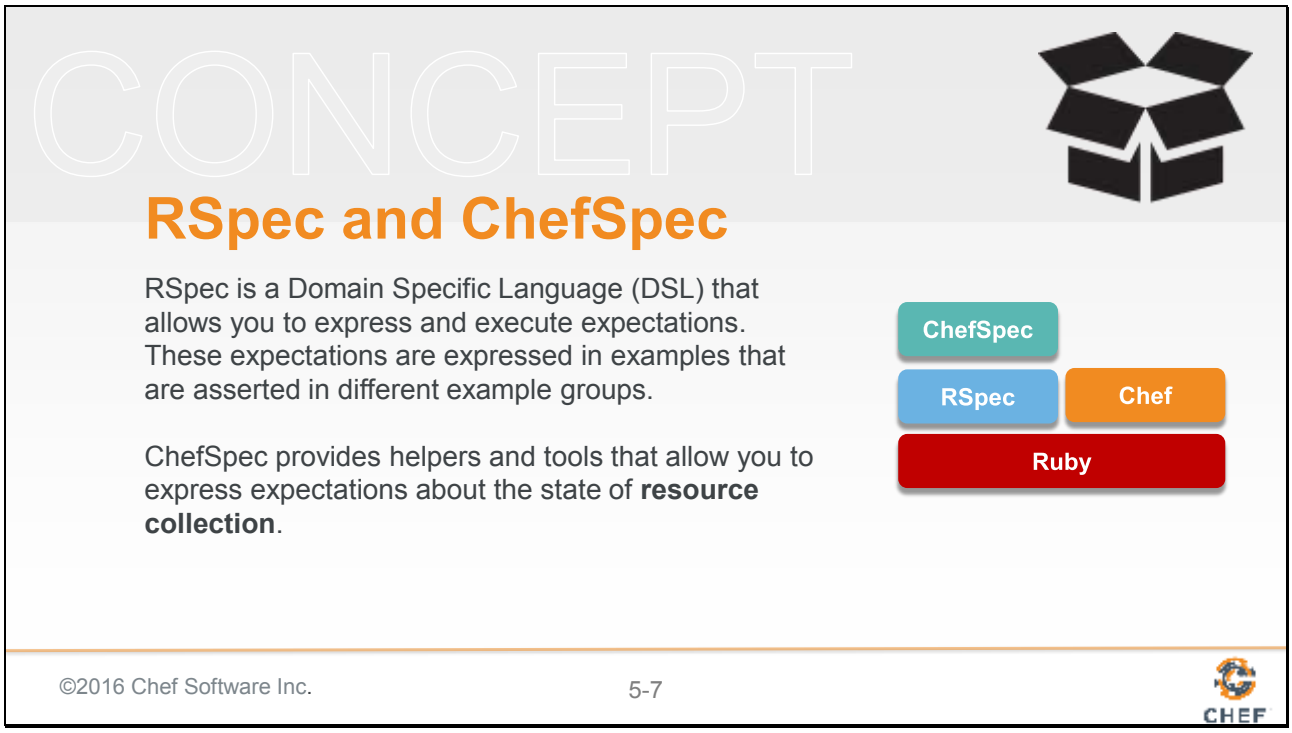


But first let's talk more about the 'Resource Collection' ...

After a cookbook and its recipes have been synchronized the majority of the cookbook content is loaded into memory by 'chef-client'. The recipes defined on the run list are evaluated during this time and the resources found within the recipes and any included recipes, are added to a resource collection. They are not immediately executed like one might assume.

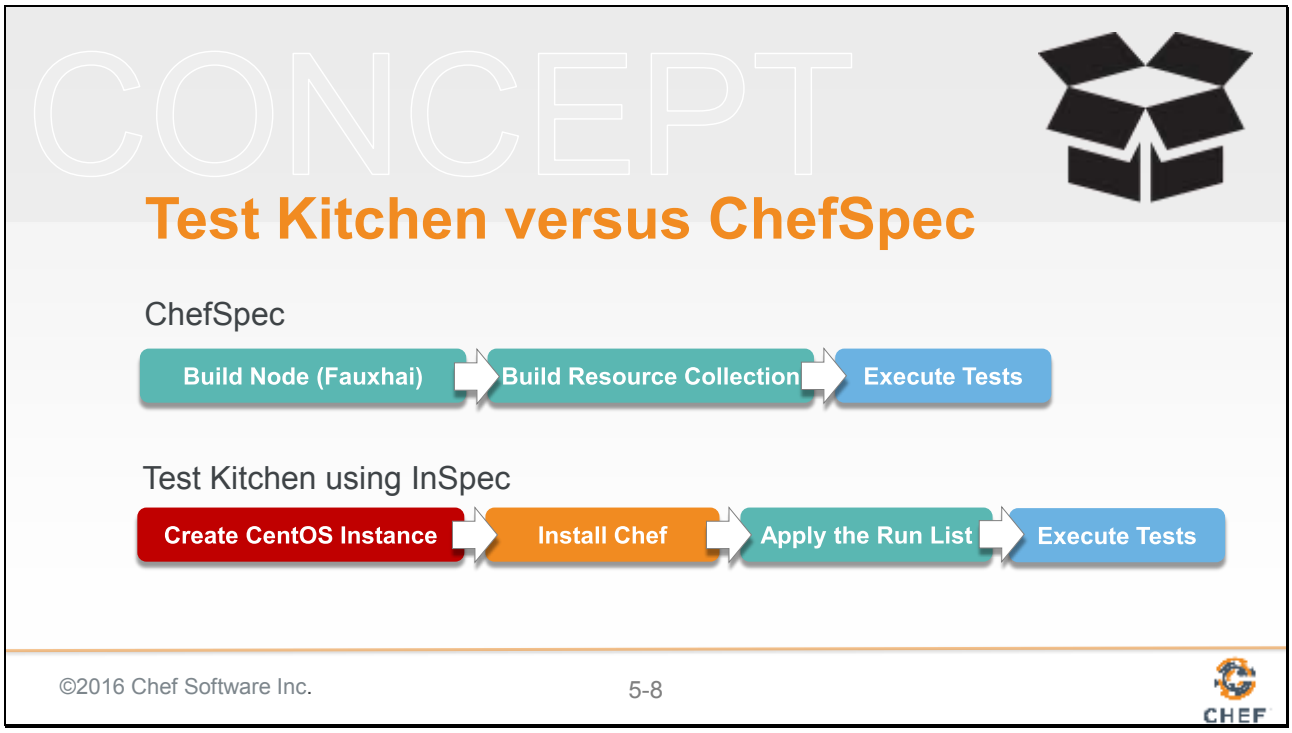
The 'Resource Collection' is almost like a to-do list for the node. It contains the list of all the resources, in order, that need to be accomplished to bring the instance into the desired state. Later, in the converge step, the resources defined in the Resource Collection are executed and perform their various forms of test-and-repair to bring the instance into the desired state.

Slide 7



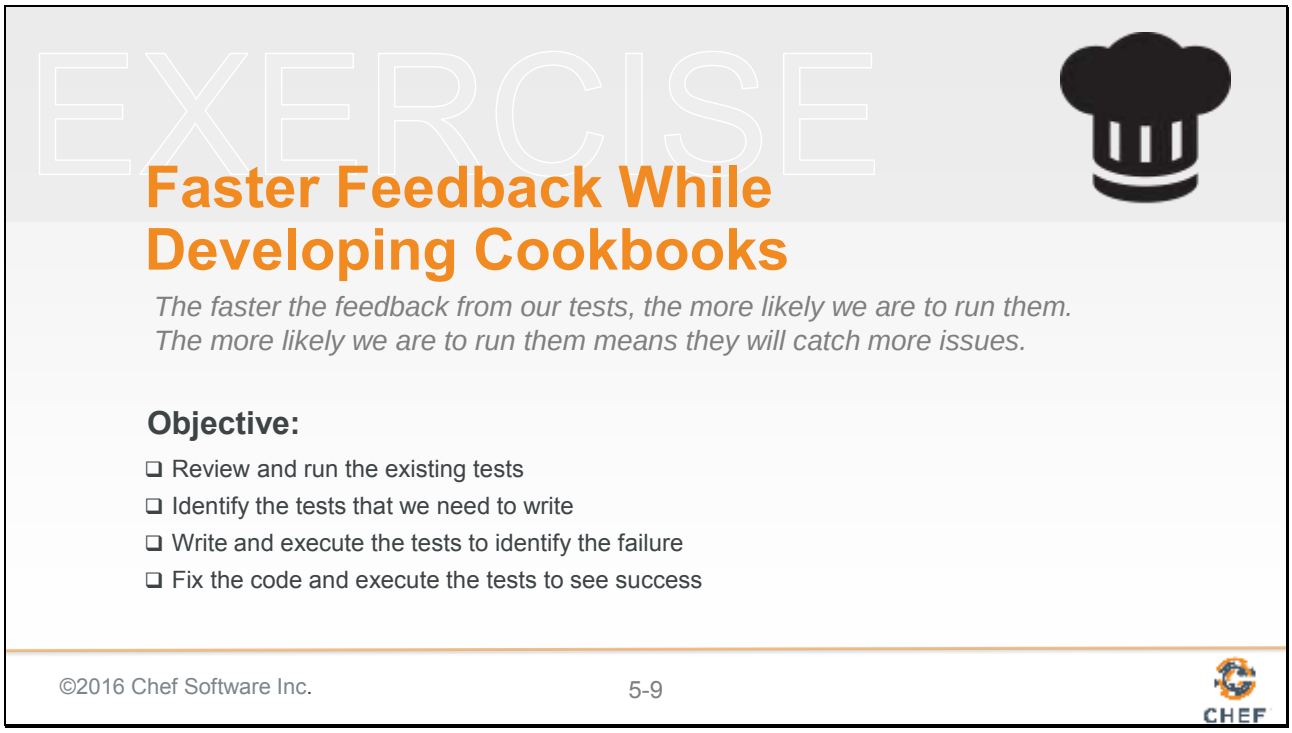
ChefSpec provides a method for us to create an in-memory execution of applying the run list, building the resource collection, and then setting up expectations about the state of the resource collection. ChefSpec, similar to InSpec is built on top of RSpec; relying on it to provide the core framework and language. The benefit to us is that a lot of the same language constructs are employed.

Slide 8



Verifying the resource collection with ChefSpec requires far fewer external dependencies and that allows us to get feedback faster but at the cost of not applying the recipes we write against a test instance. This opens us up to situations were we could compose recipes and execute examples that are shown to work because they were correctly added to the resource collection but fail when it comes time for the recipes to apply the desired state against a test instance.

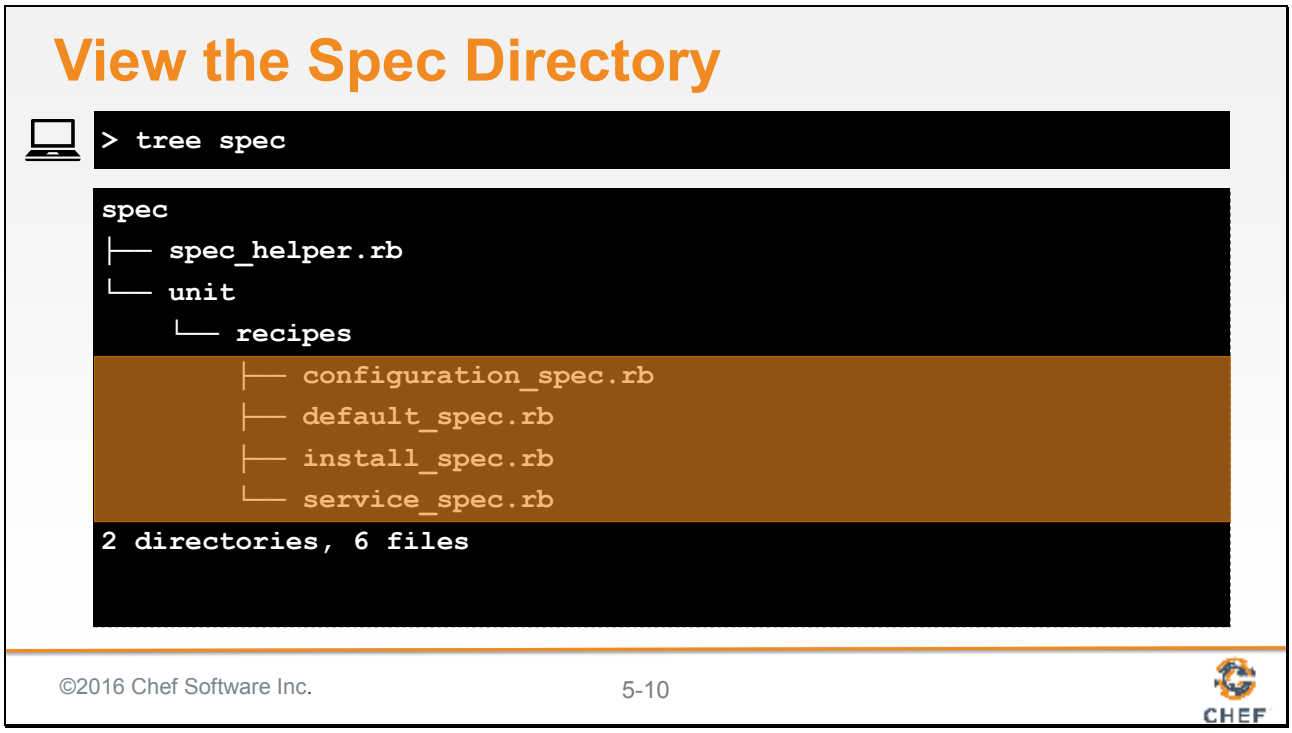
Slide 9



We have the integration test, the one defined in InSpec, executed through Test Kitchen to ensure the recipes we write behave as we expect on the test instances we define. The benefit of writing tests focused around the Resource Collection will allow us to gain feedback quickly and build a better development workflow.

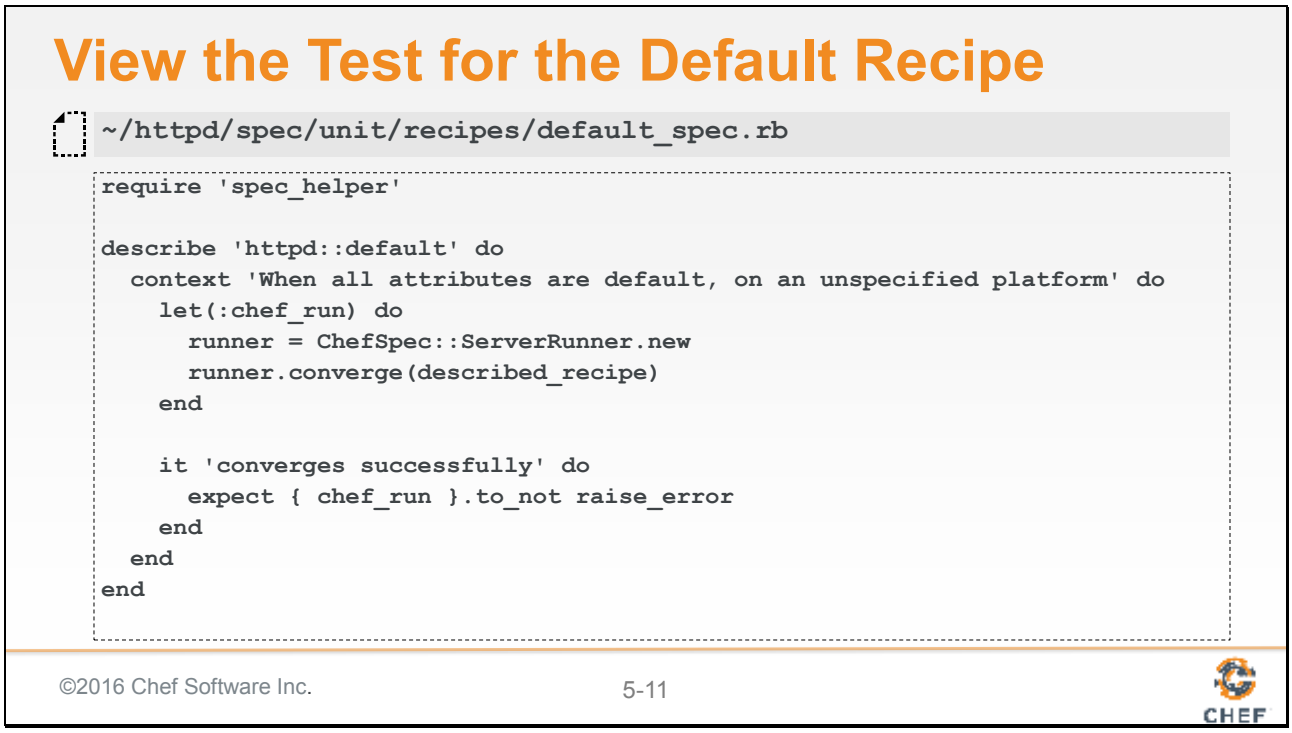
This next group exercise we will review the existing ChefSpec specifications defined for us and how we can expand them to capture our additional expectations about the Resource Collection.

Slide 10



When generating recipe files we were also given a matching specification file in the 'spec/unit' directory. The ChefSpec defined specifications are all contained within this directory.

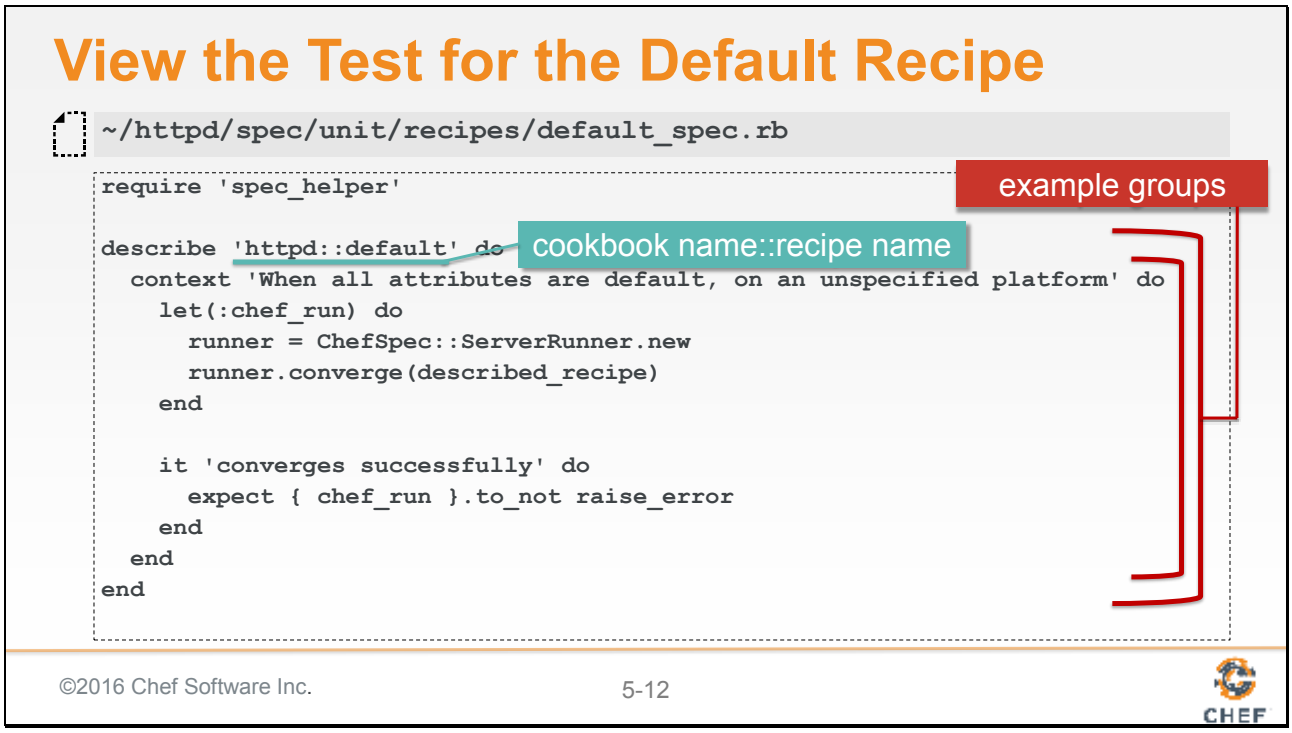
Slide 11



Open up the default specification file and lets read through and begin to understand the initial expectation that is automatically defined.

The expectations defined in this initially generated specification file should look a little familiar. This is because ChefSpec is built on Rspec. Similar to how InSpec is built. ChefSpec requires a little more setup as we are creating an in-memory execution.

Slide 12

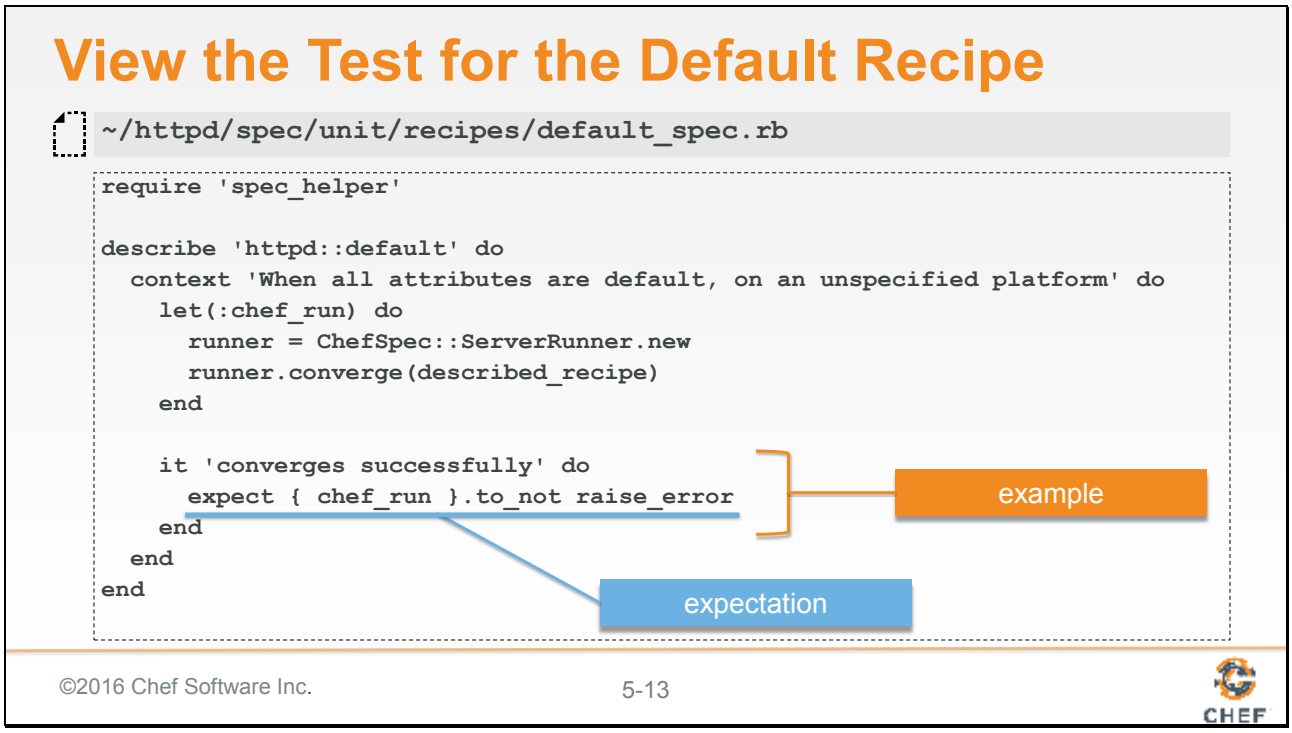


It is often common for specification files to share similar functionality. As your suite of examples grows you will often move common, shared expectations and helpers to a common file that is required here at the top of the file. This will load the contents of the 'spec_helper' file found within the root of the 'spec' directory.

ChefSpec employs RSpec's example groups to describe the cookbook's recipe. This is stating that the examples we defined within this outer example group all relate to the httpd cookbook's default recipe. Within this example group we see another context that is defined. This time using the method 'context'. 'context' and 'describe' are exactly same in almost every way. A lot of developers like to use context as it more clearly states that the example group is focused on a particular scenario. In this instance the particular scenario we are going to be specifying examples in a scenario where all the attributes are default on an unspecified platform.

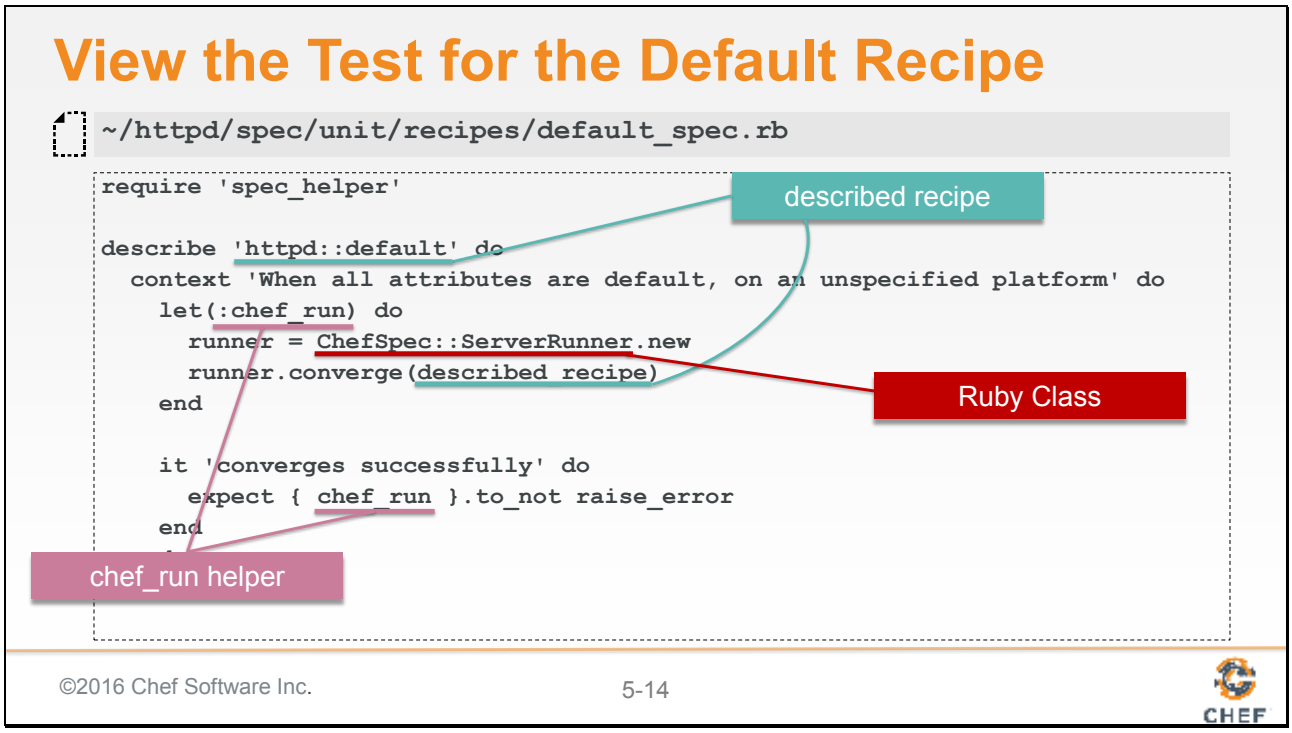
Instructor Note: 'describe' and 'context' are almost completely interchangeable with one exception. 'context' cannot be used as the outermost example group.

Slide 13



Within the inner context we finally set the stage for us to define our examples with their expectations. There is a single example defined and that is stating that when the chef run evaluates and creates the resource collection it should do so without raising an error. A situation that might raise an error is if we included a recipe that does not exist or if we were to use a resources type that does not exist.

Slide 14



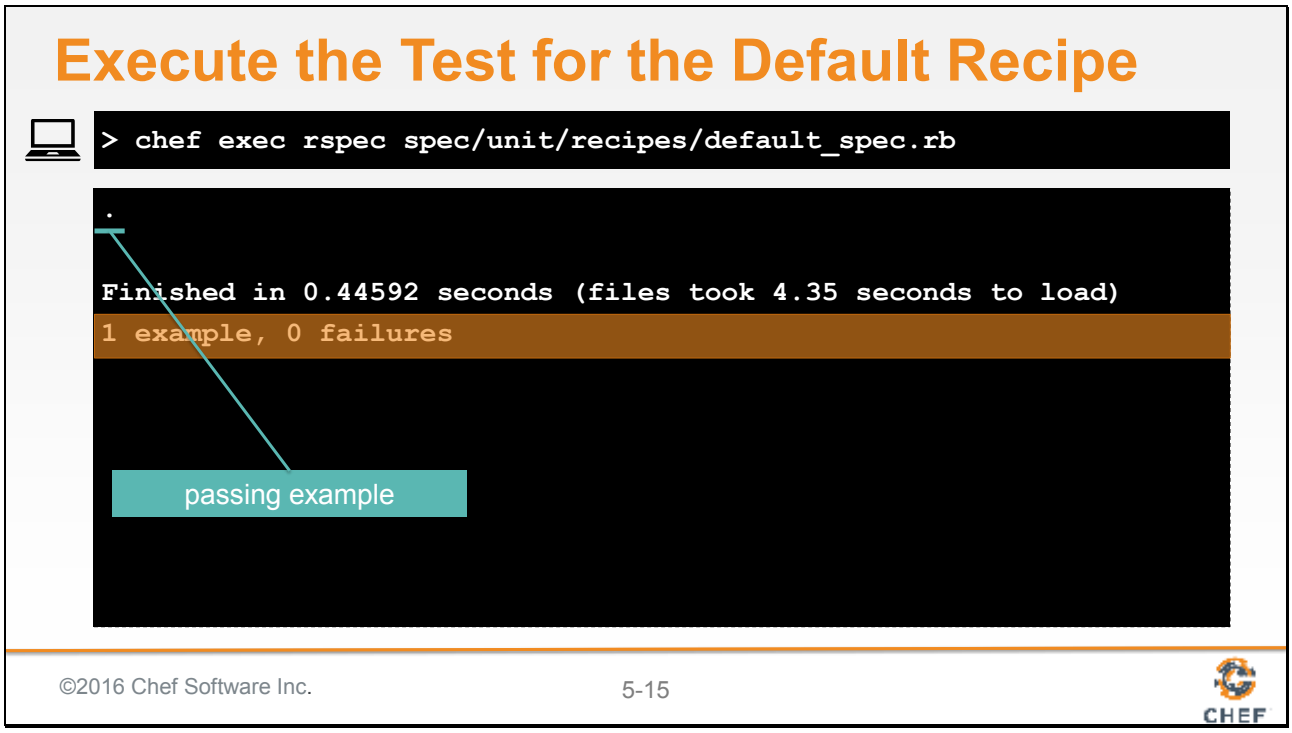
The 'chef_run' helper there is being provided by the 'let' defined above the example within the same context. Defining the 'chef_run' in the 'let' above is done with a Ruby Symbol. This is simply naming it so that it can be used within any of the examples in the current context and even sub-contexts. The helper is simply executing some code that sets up an in-memory chef-client run with a Chef Server.

The 'ServerRunner' is a class defined within the 'ChefSpec' namespace. All Ruby classes have the method 'new' which will return an object which is a new instance of that described class. The object is stored in a local variable, named 'runner', which immediately invokes a method 'converge' with a single parameter 'described_recipe'

The parameter 'described_recipe' refers to the recipe defined in the outermost describe. This is mostly for convenience so that we do not have to redefine the same String multiple times within the same specification file.

The goal of this single, boilerplate example is very simple: perform a chef-client run and ensure there are no errors. Now, let's execute this specification.

Slide 15

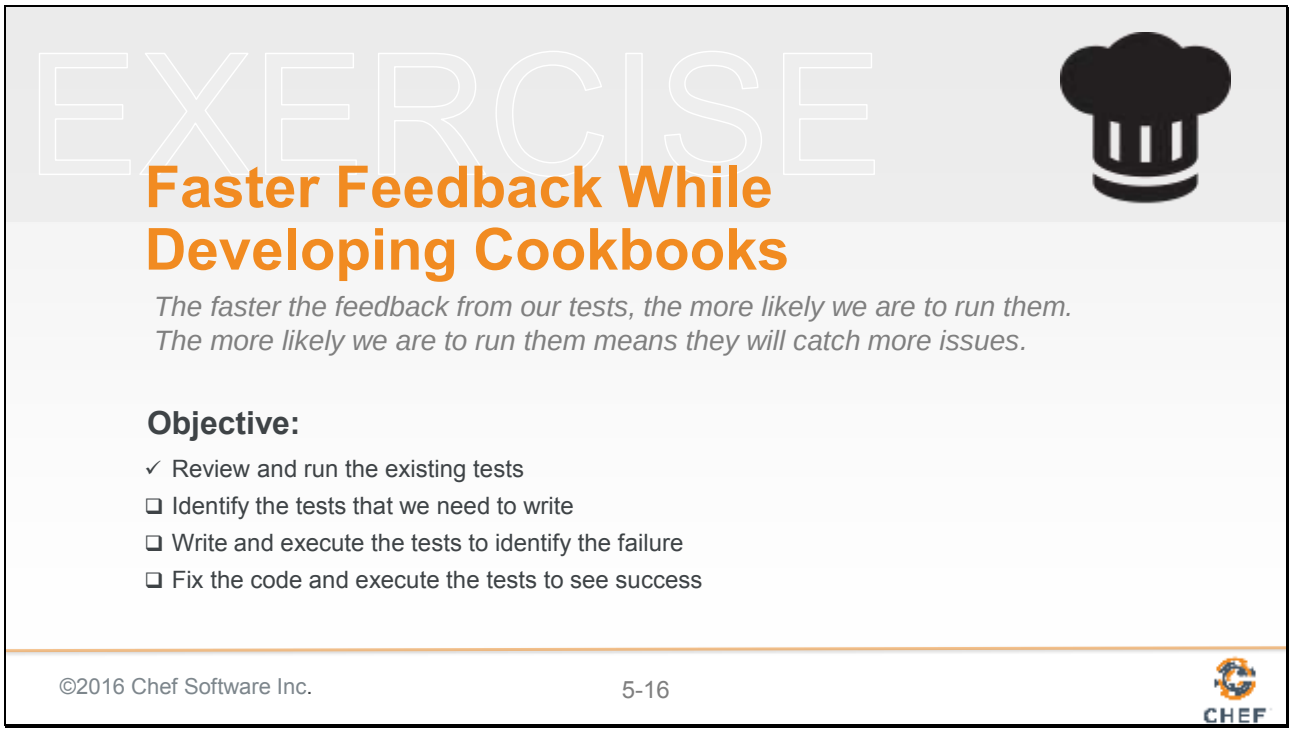


To execute the specification file defined you will need to run the command 'rspec'. The 'rspec' command was installed with the Chef Development Kit (ChefDK) on the workstation. It is contained in an additional folder of tools embedded within the ChefDK that are not added to the system path. This is because some Chef developers are Ruby developers and may already have a version of RSpec installed. Specifying the 'chef exec' as a prefix loads the context of all these embedded tools and allows them to be executed on the command-line.

The 'rspec' command accepts many parameters. The most important one is used here and that is specifying the file path to the specification we want to execute. When the command executes a summary of the executed examples will be displayed at the bottom. At this moment it looks like the one expectation completes successfully. The chef run completes without any errors.

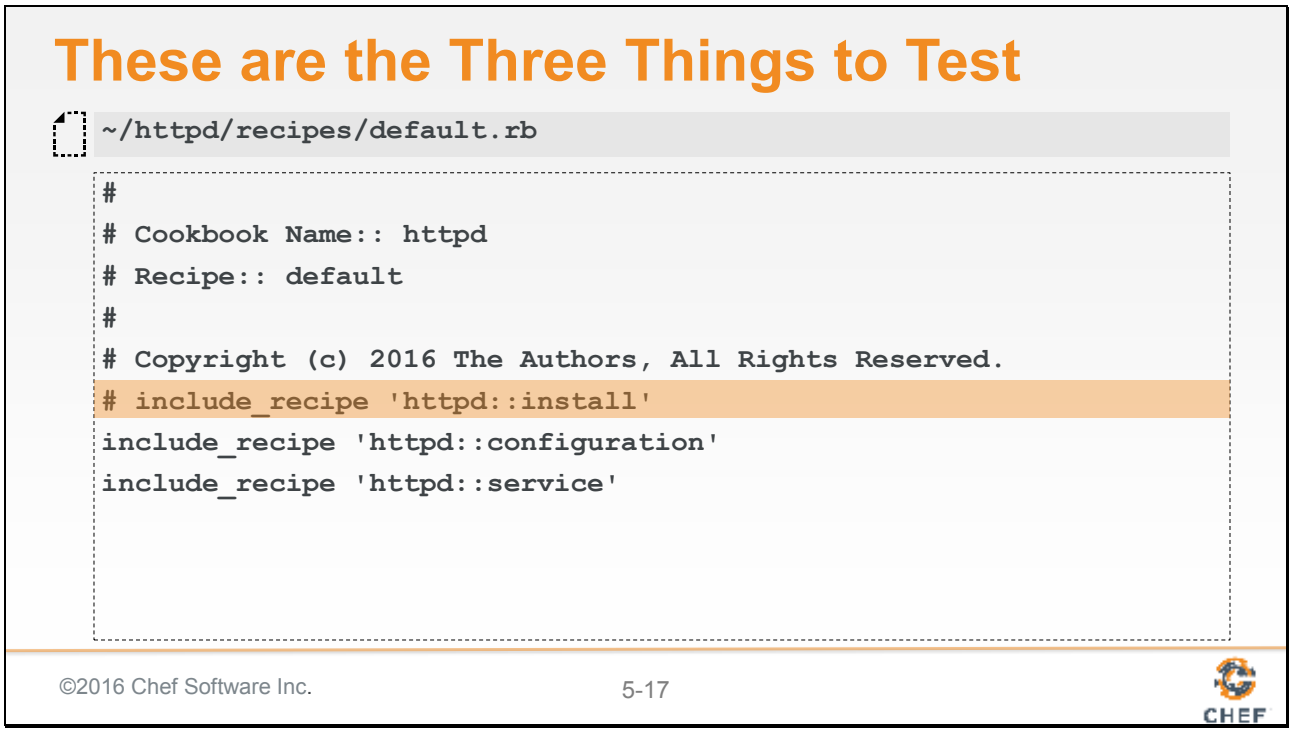
Instructor Note: On the workstations the learners do not need to prepend the rspec command with 'chef exec'. This is because 'rspec' and all the other tools embedded in the ChefDK have been added to the path. On a learner's local system this is likely not the case and so they will need to type this entire command with prefix.

Slide 16



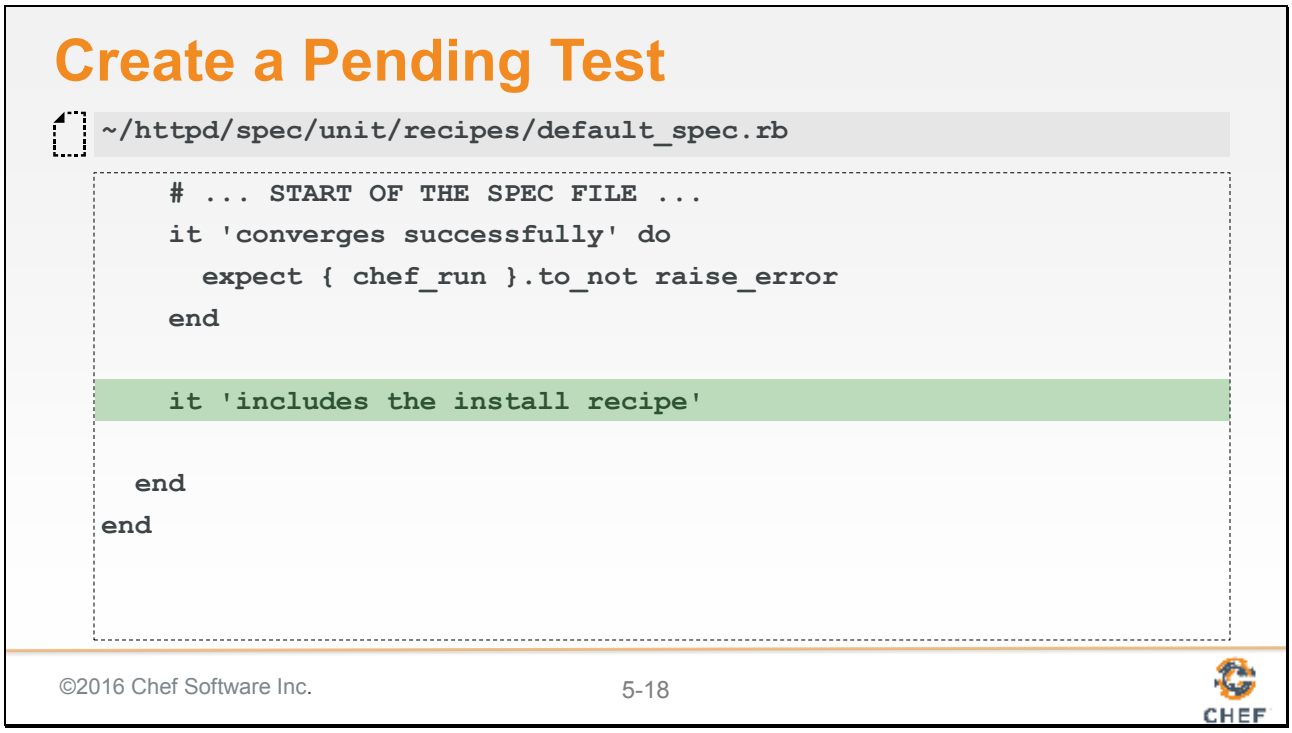
We have the language and the tool that will allow us to express our expectations. We now need to examine the recipe again to see what example or examples we want to define within the specification file.

Slide 17



Within the default recipe we commented out the line that included the install recipe from the httpd cookbook. This seems like an expectation that we want to define. When converging the default recipe we expect that the install recipe from the httpd cookbook would be included.

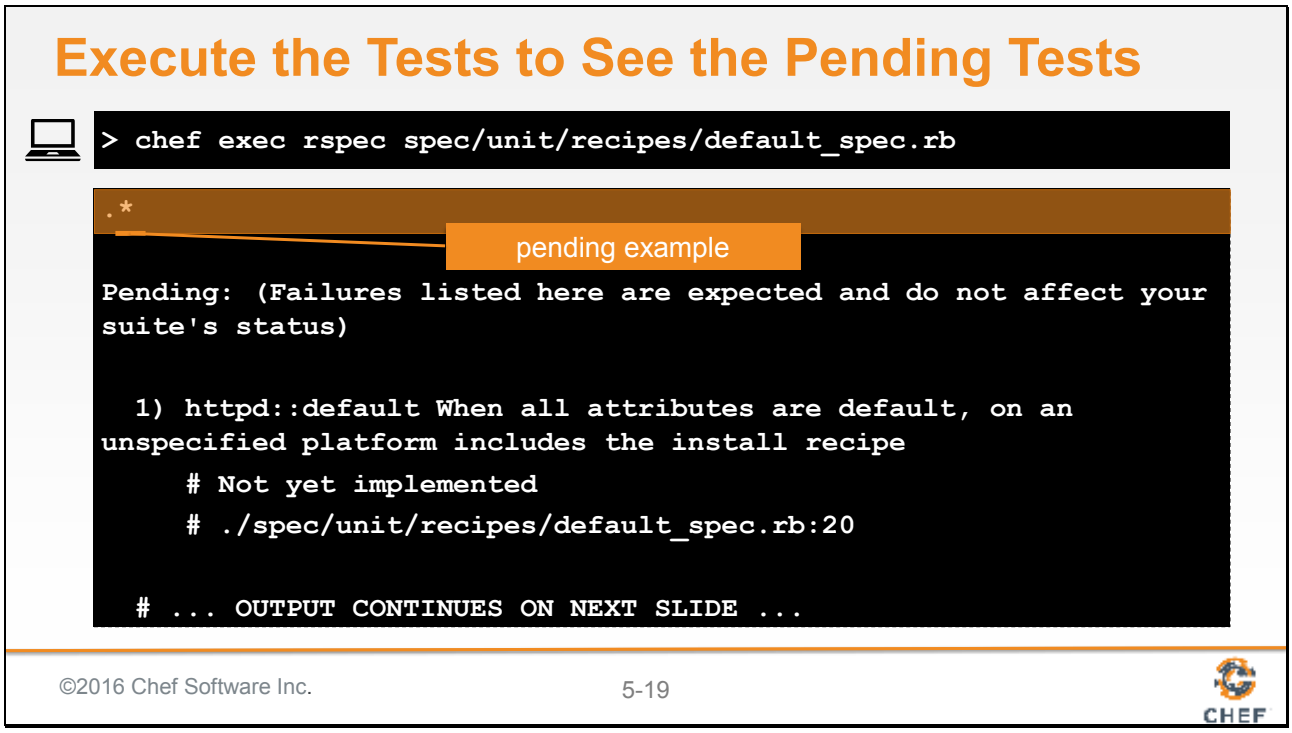
We do not yet know how to define this expectation but we know the work that we want to accomplish. So lets take this one step at a time then and first capture the description for the example even if we do not yet know how to express the expectation.



Returning to the specification we can describe the example that we want to create without having to know how to define the expectation by defining an example without the block. RSpec treats these examples without the associated block as a pending test.

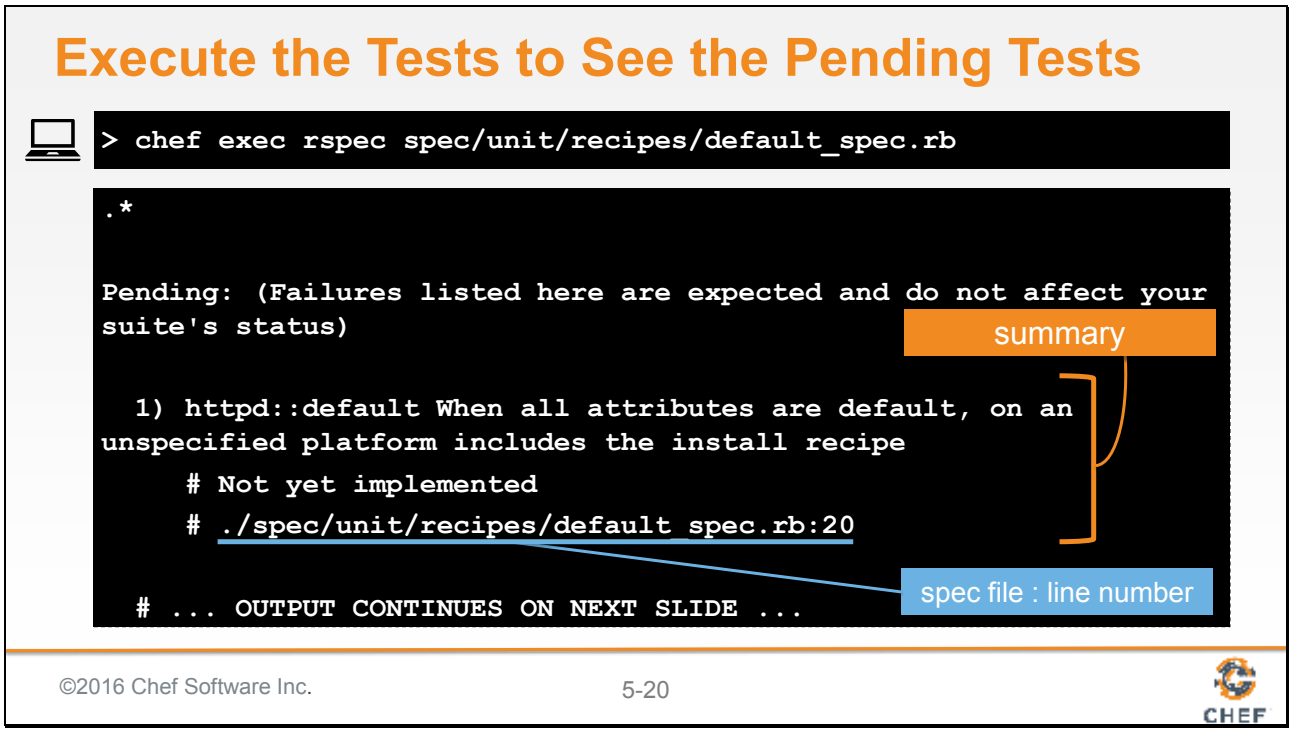
This is an incredibly useful feature when you want to start expressing your examples. This allows you to quickly identify all the examples without getting mired in the details of their implementation.

Slide 19



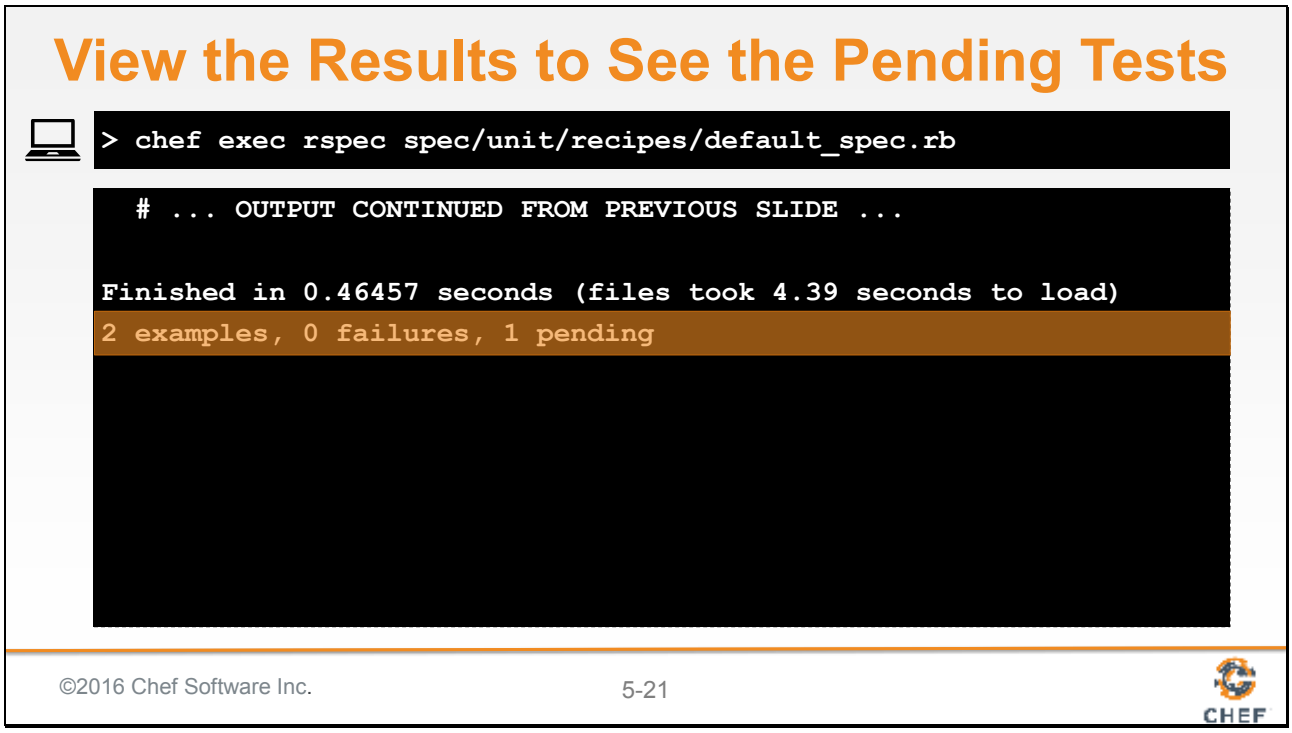
When executing 'rspec' again we should see the new pending example that we defined within the specification file.

Slide 20



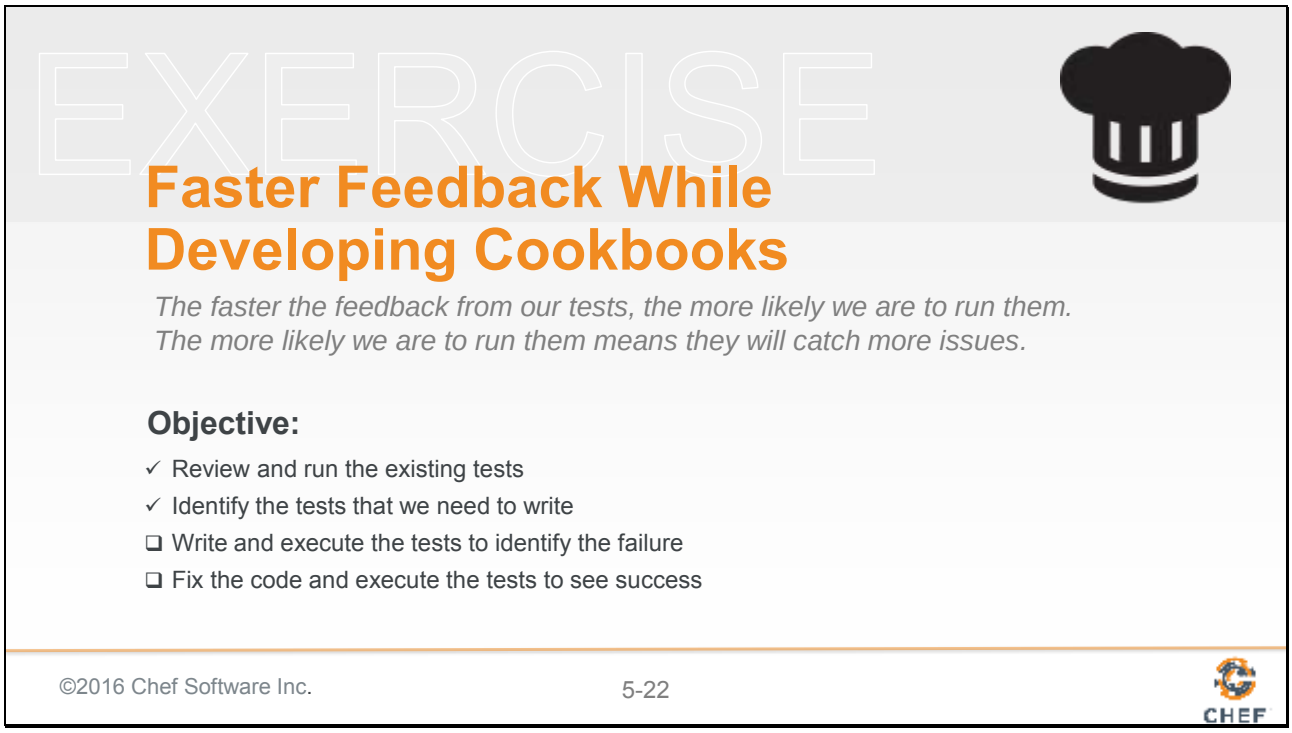
RSpec's pending summary is similar to the failure summary. The pending examples are identified and then finally they are collected together in list. Each pending example will show the words you used in the description text in a single sentence. Below that it will state the example is not yet implemented and then finally display the file path and line number of where it can be found.

Slide 21



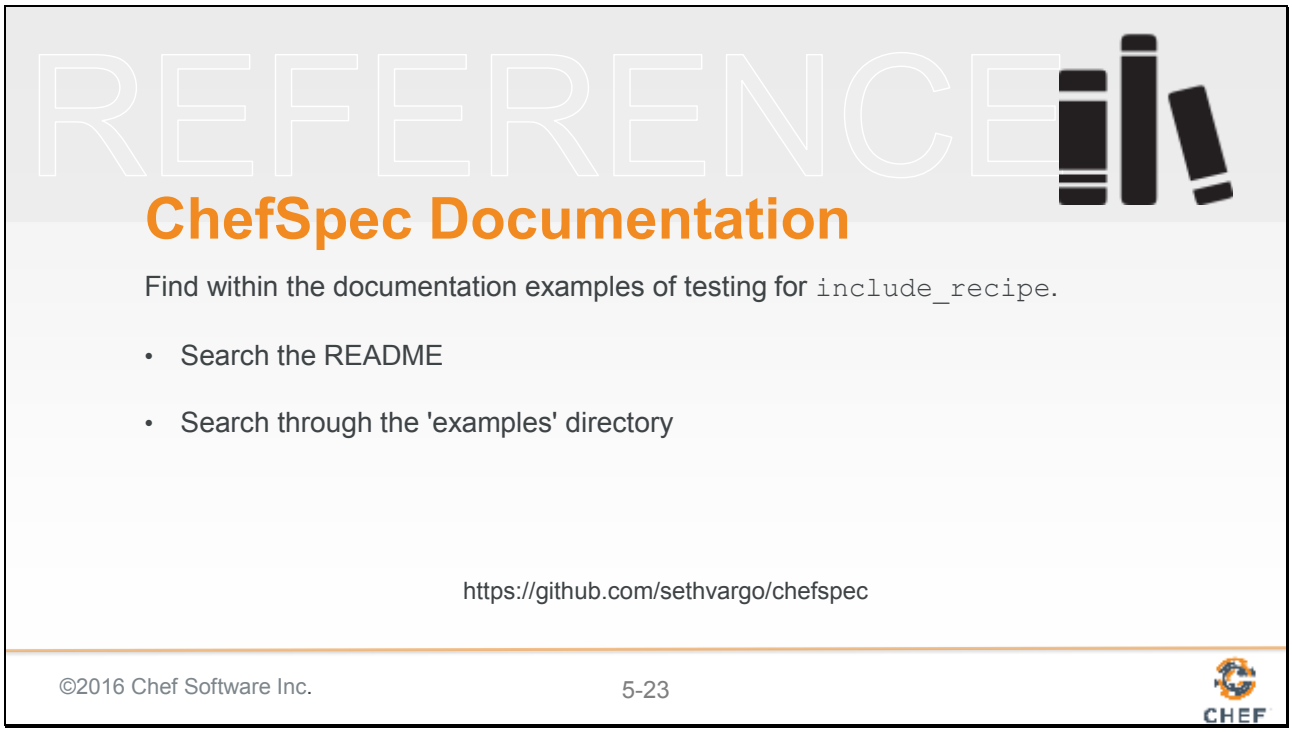
The summary will now display that an additional example has been added and it will be reported as being set to pending.

Slide 22



Now that we have defined the pending example, setting up the work for ourselves, it is time to learn how to express the expectation.

Slide 23



To understand how to express an expectation we need to go to the documentation. The ChefSpec README provides a wealth of examples in the README. In the past an 'include_recipe' example has been one of the many examples shared in the README. Use the search feature of your browser to find it within the document.

If it is not there, the ChefSpec project has a top-level folder named 'examples' which contains examples for nearly every feature that ChefSpec is able to define expectations. Searching through there you will find a folder titled 'include_recipe', within it should a folder the shows the recipes and the matching specifications.

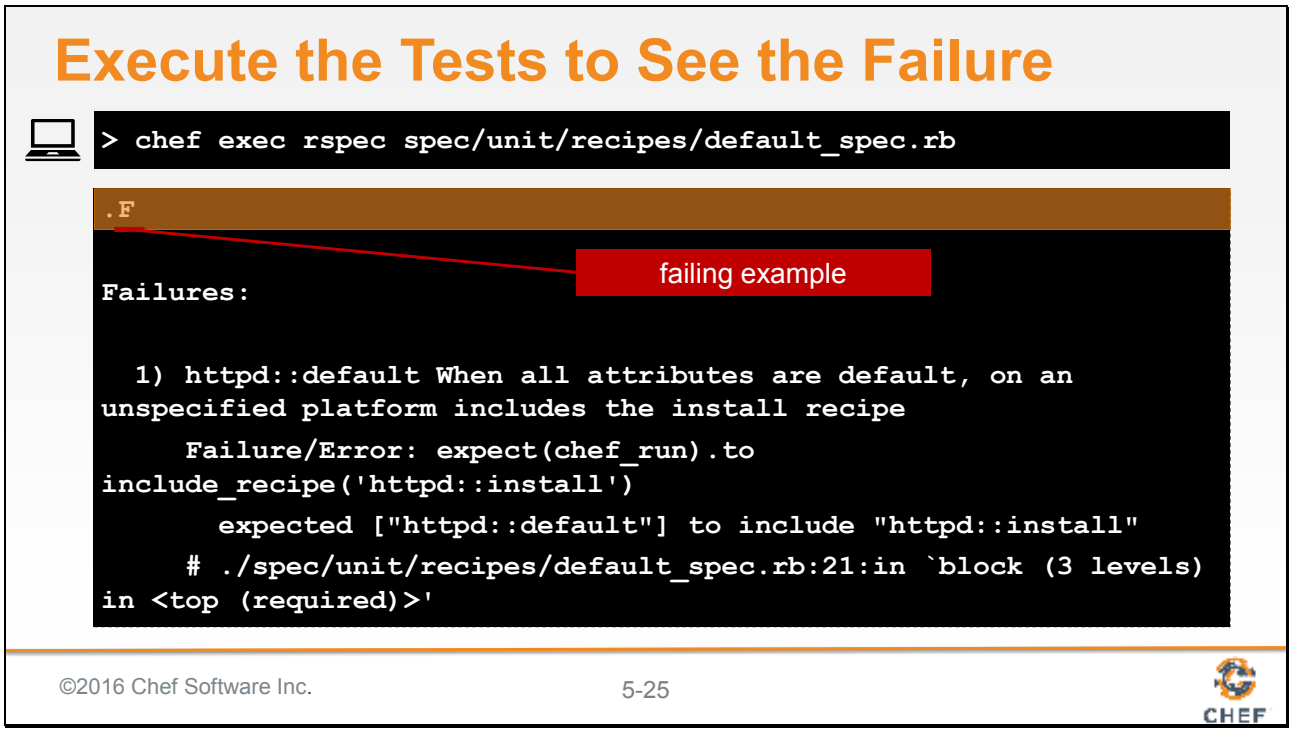
Write the Test that Verifies the Include Recipe

~/httpd/spec/unit/recipes/default_spec.rb

```
# ... START OF THE SPEC FILE ...
    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end

    it 'includes the install recipe' do
      expect(chef_run).to include_recipe('httpd::install')
    end

  end
end
```


Returning to the specification file we now need to expand the example to include the expectation we want to write. To do that we add a 'do' to the end of the example. We move to the next line, indent two spaces and then define the following expectation. The expectation uses a natural language way of expressing the expectation. Here we are expressing the expectation that the 'chef_run' includes the recipe with the specified name.

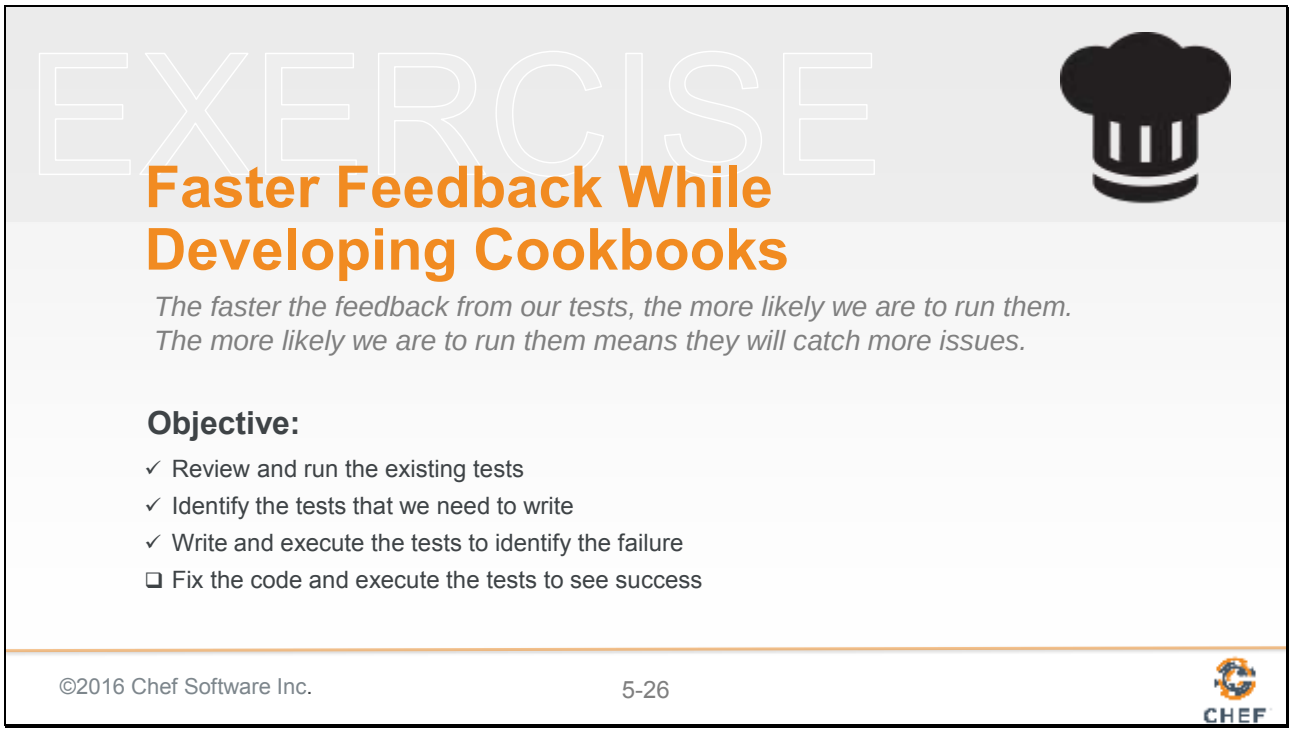
Slide 25



With the example defined with the expectation when we execute 'rspec' we see the failure that eluded us we ran 'kitchen converge & verify' on an existing very quickly.

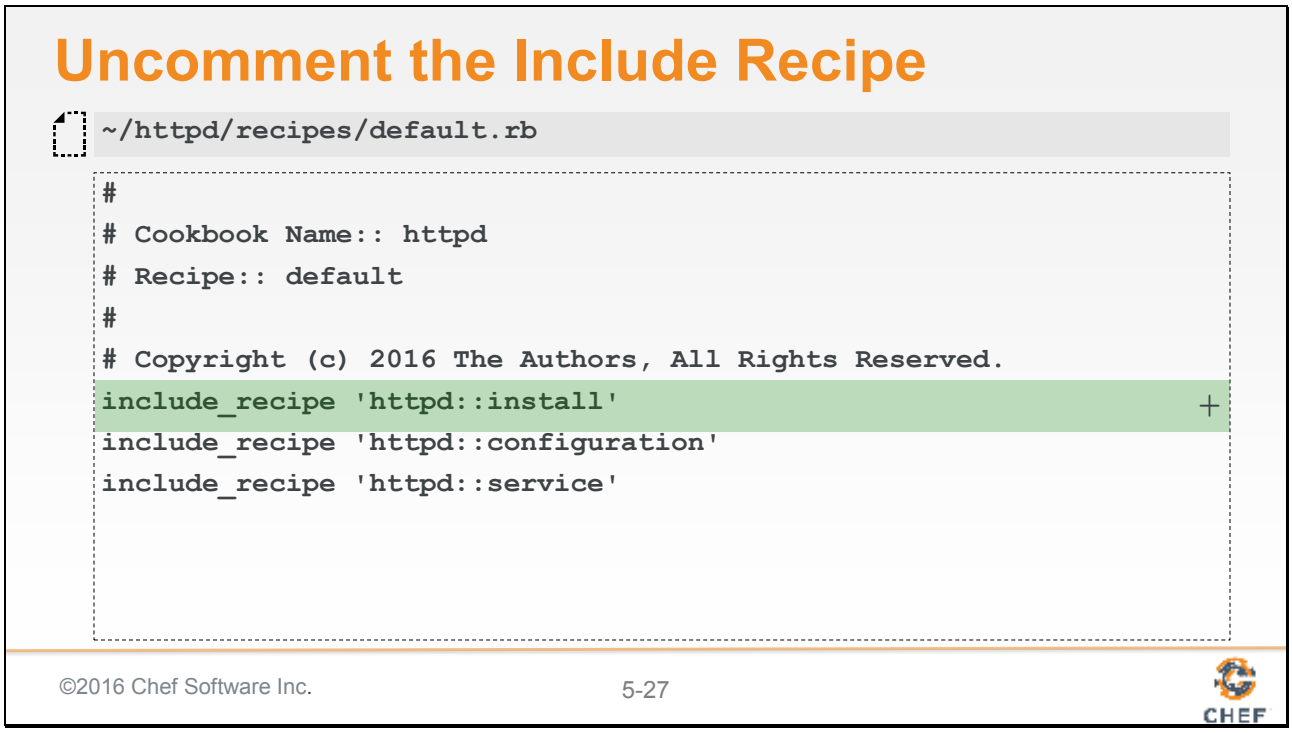
The failure summary here is similar to the failure summary return by RSpec when employed by Test Kitchen. The example is displayed, the expectation is expressed, the failure to meet expectation and file name and line number within the file where to find the expectation.

Slide 26



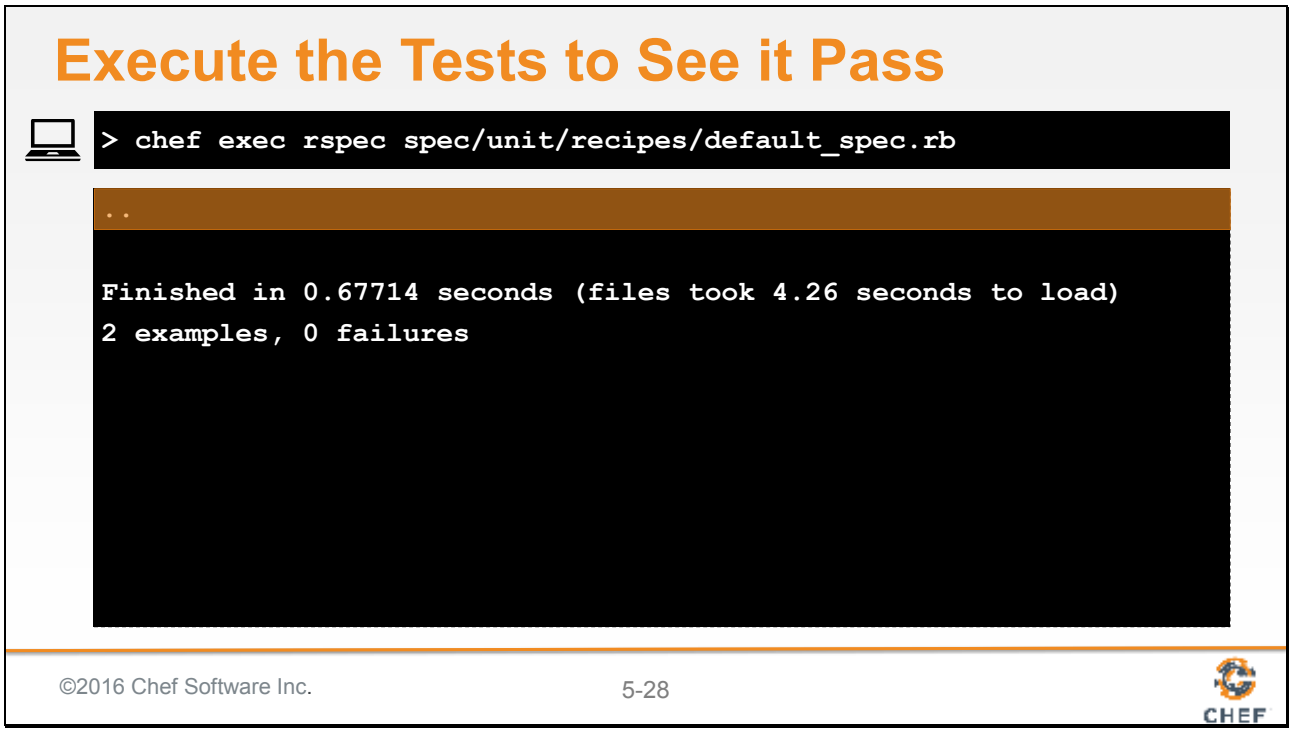
Now that we have a failing test it is time to fix the problem.

Slide 27



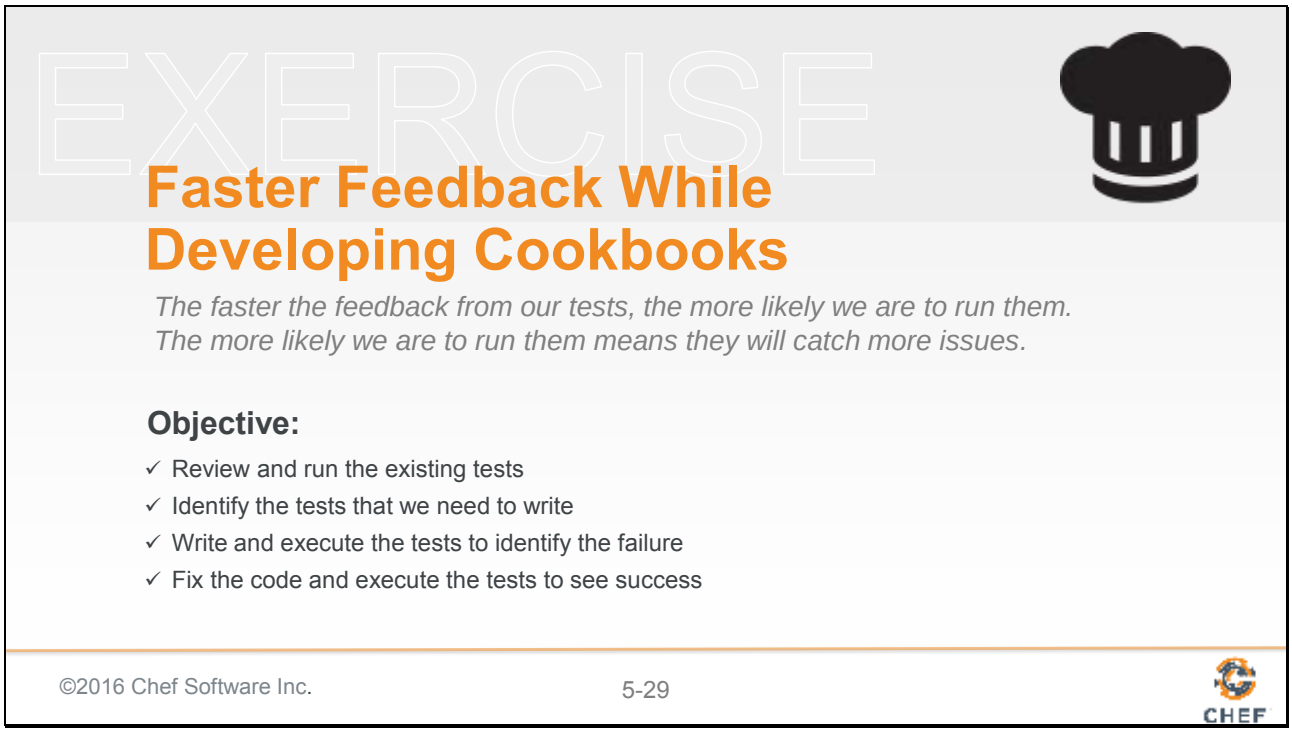
Returning to the default recipe it is time to restore the code that we previously commented out.

Slide 28



Executing 'rspec' one more time should show the previous failing example now as a passing example.

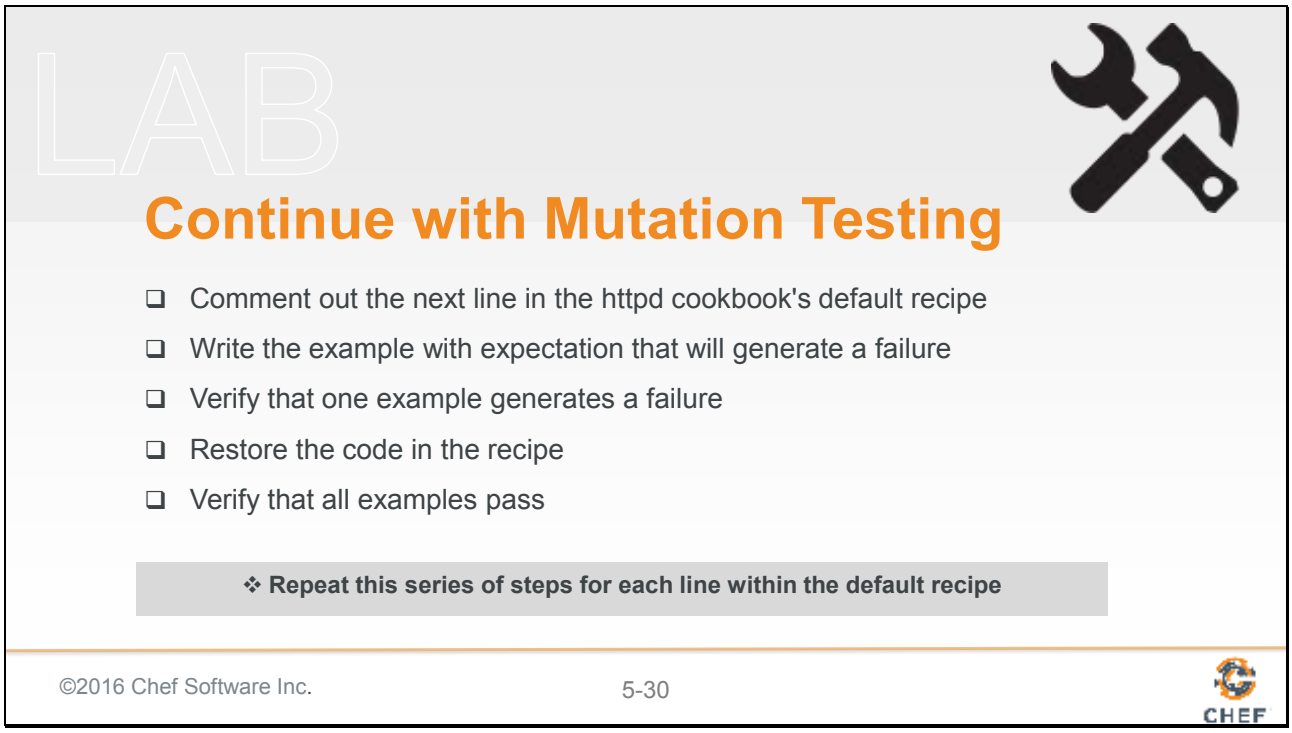
Slide 29



Now we can confidently state that the default recipe includes the install recipe and we can receive this verification in a faster feedback cycle then we saw with running 'kitchen test'.

Mutation testing is not Test Driven Development (TDD) but the act that we performed was fairly close. This is a tactic that is useful when you are writing expectations for already defined recipes for existing cookbooks or when it feels near impossible to start with the tests first. This process does one of the important aspects of TDD which is ensure the expectations we set correctly capture the state of the code.

Slide 30

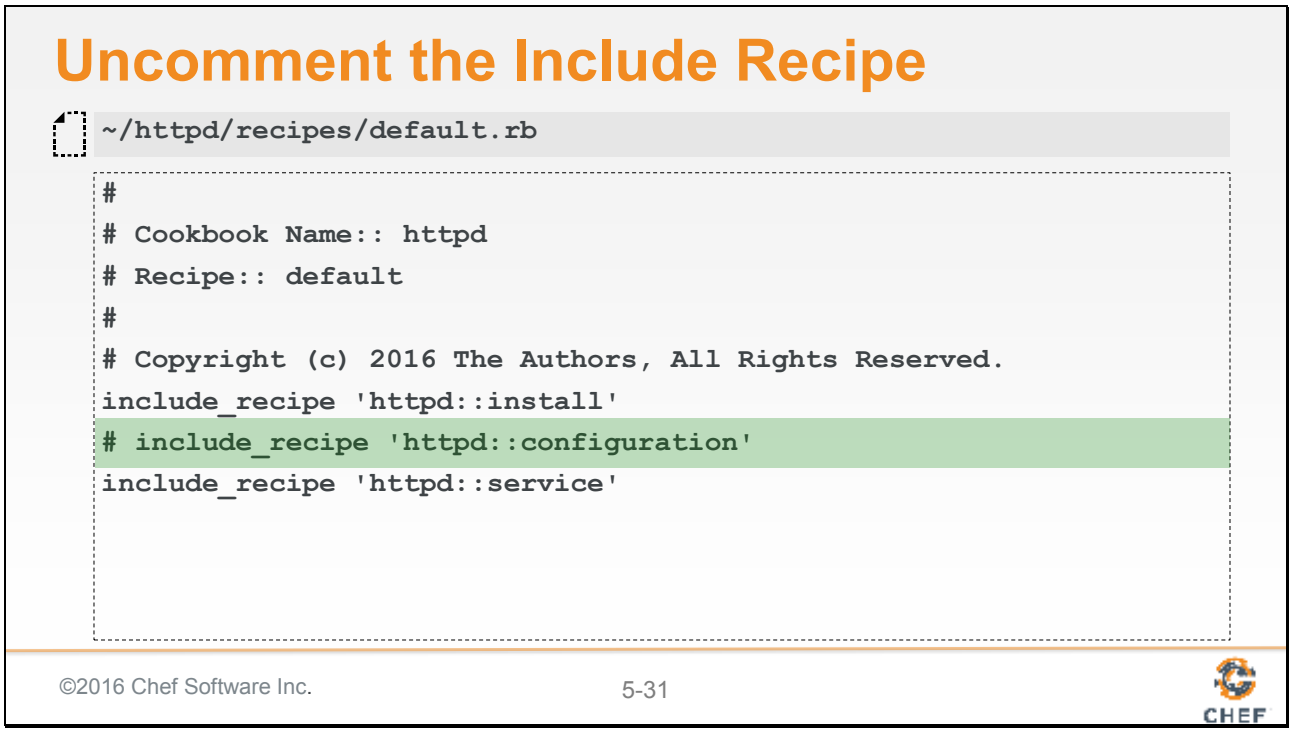


There are few more chances to reinforce this process. As an exercise continue mutating the code within the default recipe, defining the expectations, and then fixing the code. Create a single mutation at a time and become focus on understanding the process of moving between files and executing commands.

Instructor Note: Allow 10 minutes to complete this exercise

Instructor Note: The learners could accomplish both of tasks at the same time. They likely will want to do that. I would encourage you have them perform the steps separately as it will emphasize the activity of moving between the recipe, the specification file, and their shell.

Slide 31



Let's review by walking through one more example within the default recipe. Another line within the recipe is similar to the first one except it is concerned with the inclusion of the configuration recipe. Here it is commented out.

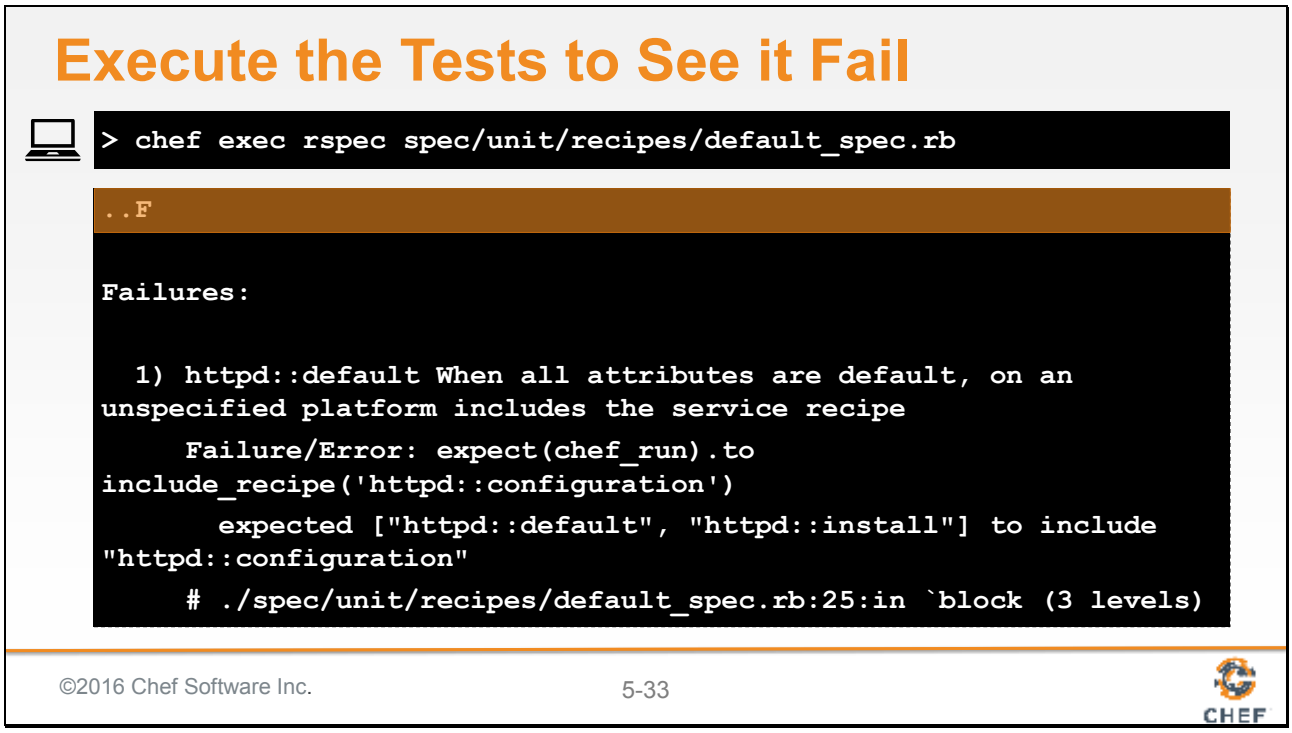
Slide 32

Write the Test that Verifies the Include Recipe

~/httpd/spec/unit/recipes/default_spec.rb

```
# ... START OF THE SPEC FILE ...

   it 'includes the install recipe' do
     expect(chef_run).to include_recipe('httpd::install')
   end

   it 'includes the configuration recipe' do
     expect(chef_run).to include_recipe('httpd::configuration')
   end

  end
end
```


Returning to the specification file to define the example and the new expectation.



Seeing the failure when executing te 'rspec' command.

Slide 34

Uncomment the Include Recipe

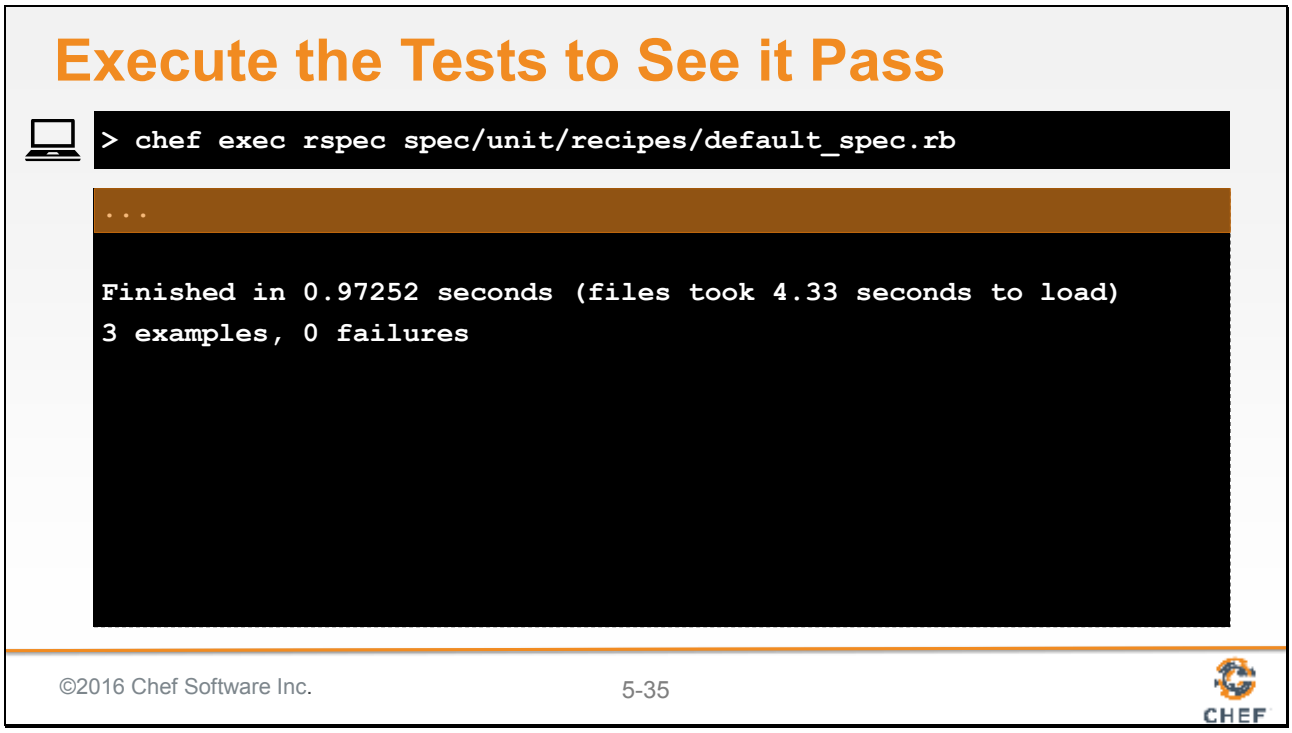
~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
include_recipe 'httpd::install'
include_recipe 'httpd::configuration'
include_recipe 'httpd::service'
```


CHEF

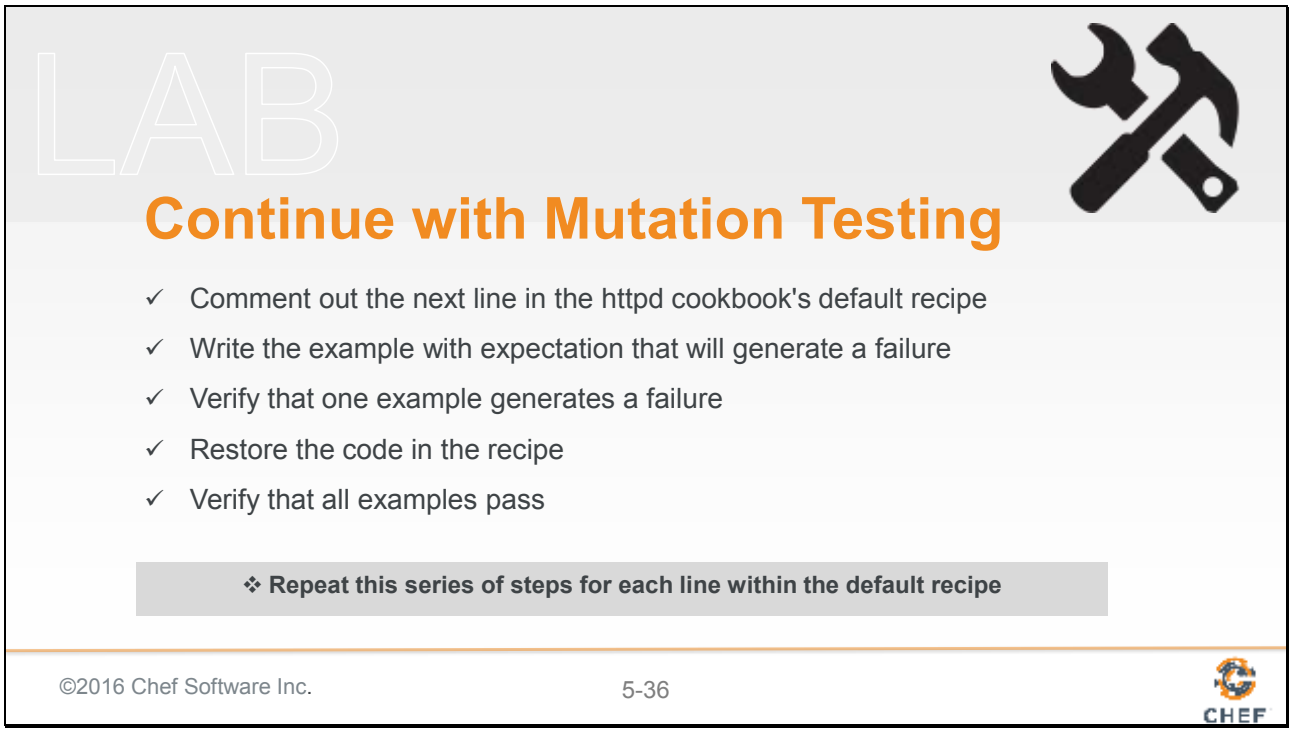
Restoring the code to its previous state

Slide 35



Executing 'rspec' again to verify that the expectations have been met successfully

Slide 36

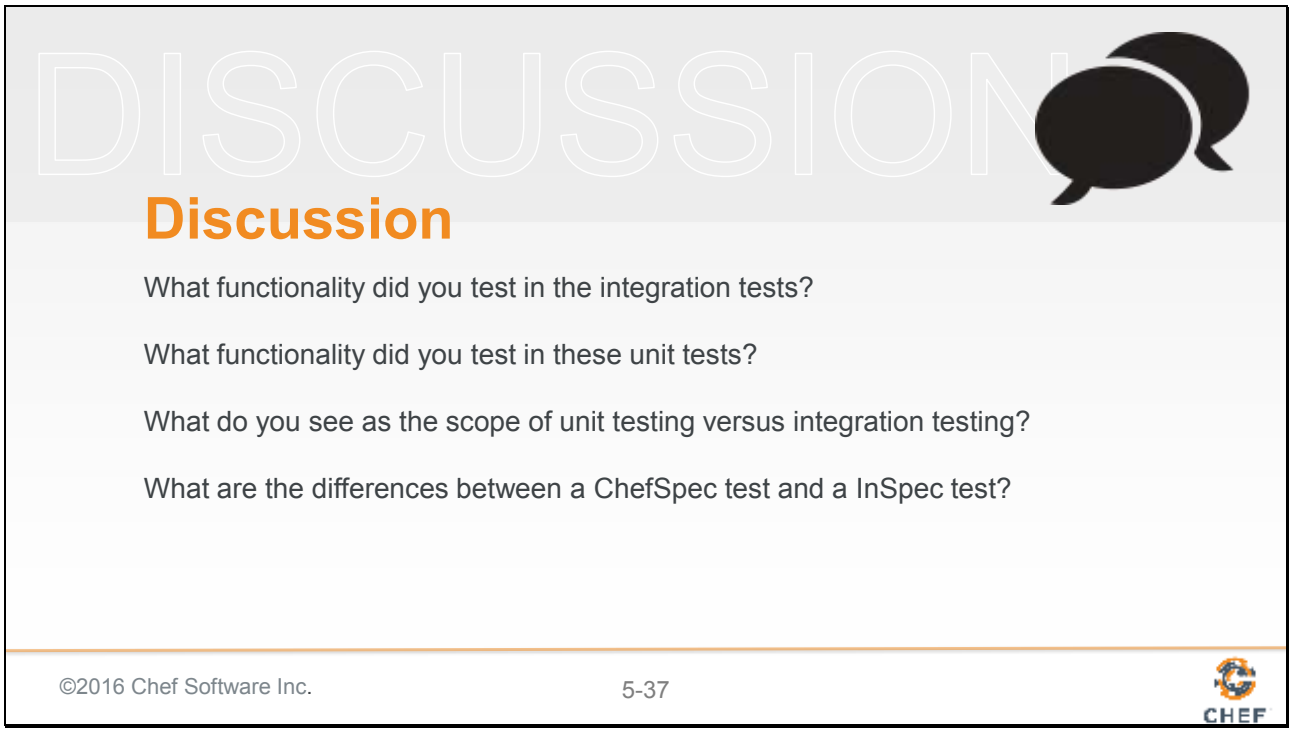


There are more mutations that you could try within the default recipe and other recipe files that exist within the cookbook but this is a good point to stop and enjoy the work that you have accomplished.

The feedback cycle on using Rspec to execute ChefSpec examples returns results faster than we saw with Test Kitchen and gives us a good understanding of what is being added to the 'Resource Collection'.

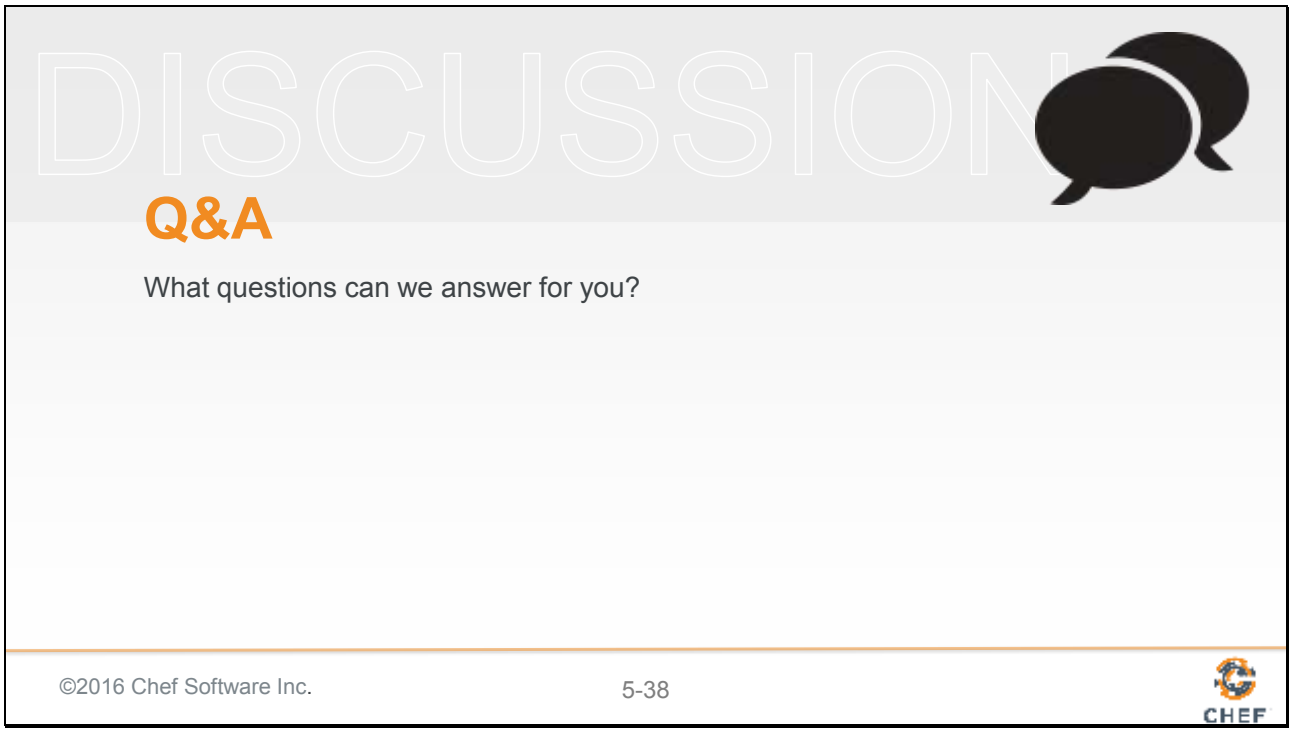
Let's have a discussion.

Slide 37



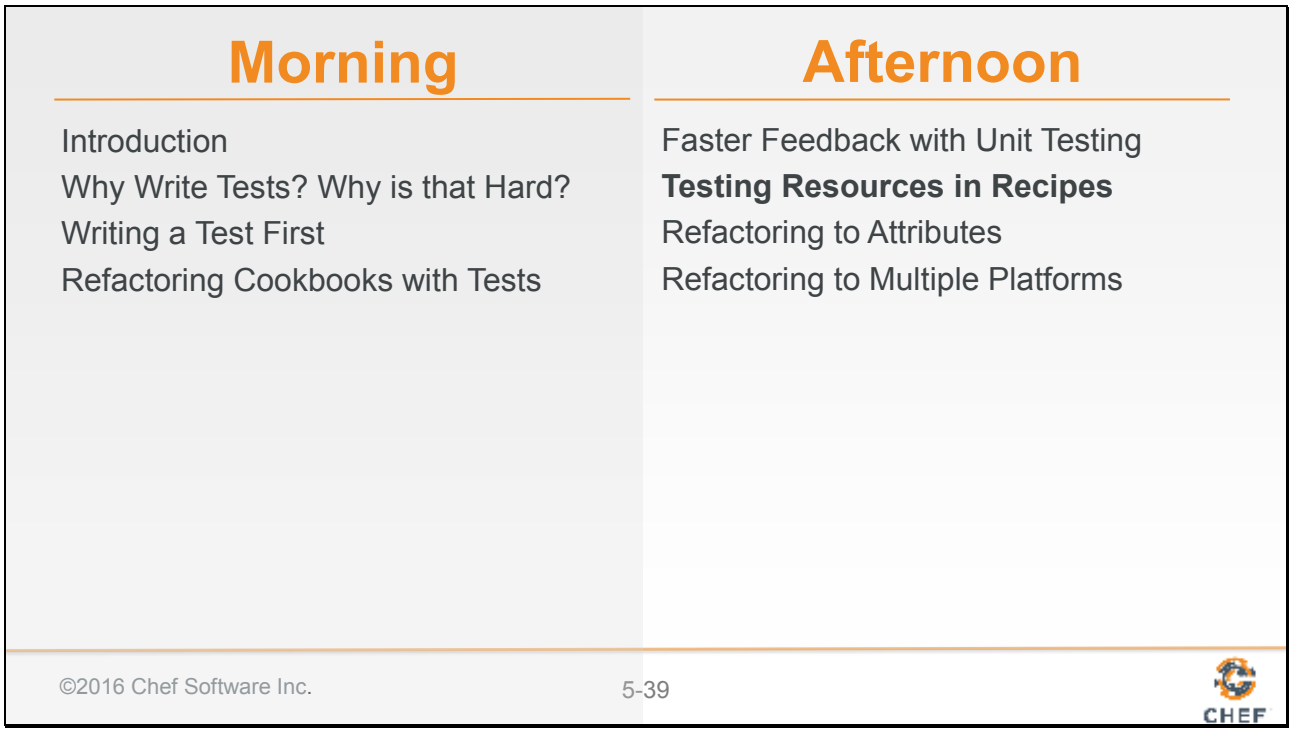
Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 38



Before we complete this section, let us pause for questions.

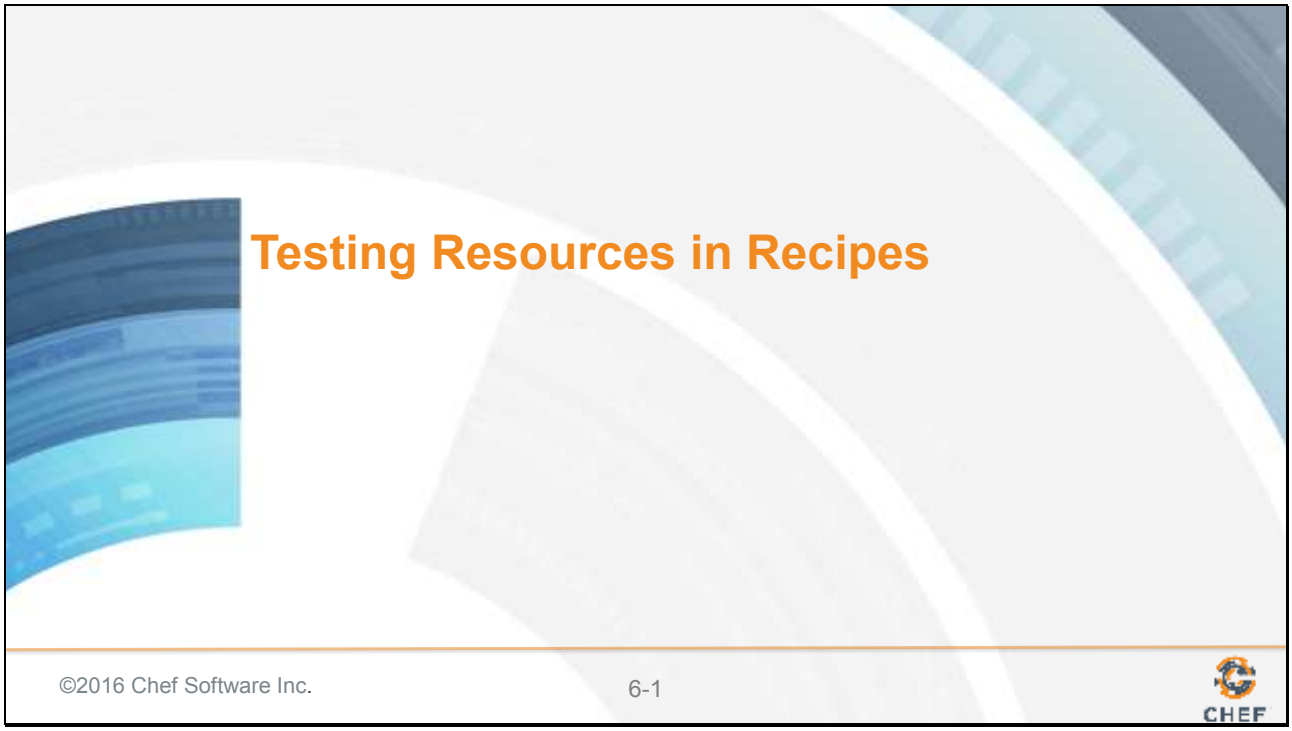
Slide 39



We have the faster feedback that we set out to create for us at the beginning of this section. We were able to verify the work being performed in the default recipe. Now it is time to focus our attention on the remaining recipes with in the cookbook and set up expectations on the resources that they define.

Slide 40

6: Testing Resources in Recipes



The default recipe we refactored moved the resources into individual recipes that will promote their ability to be composed in other cookbooks. Now its time to take a look at the resources we defined and explore writing examples to verify their state as well.

Slide 2

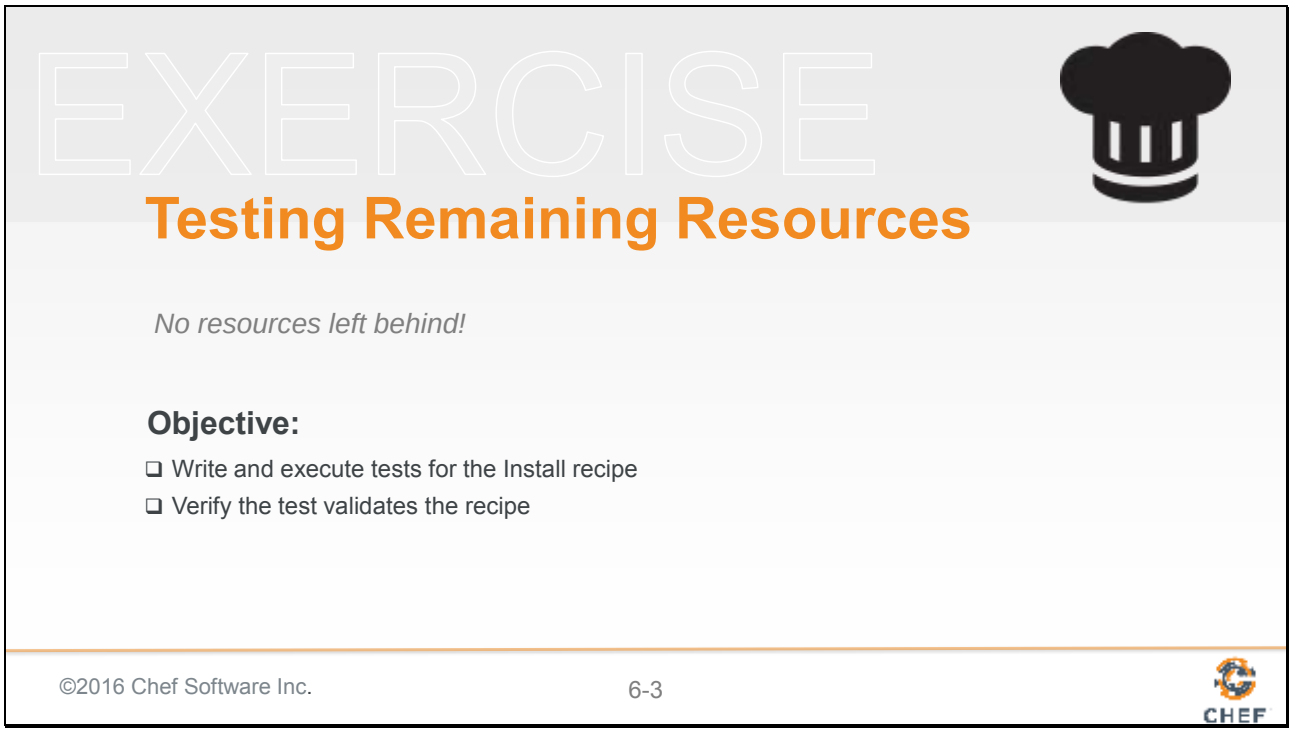
Objectives

After completing this module, you should be able to:

- Test resources within a recipe using ChefSpec

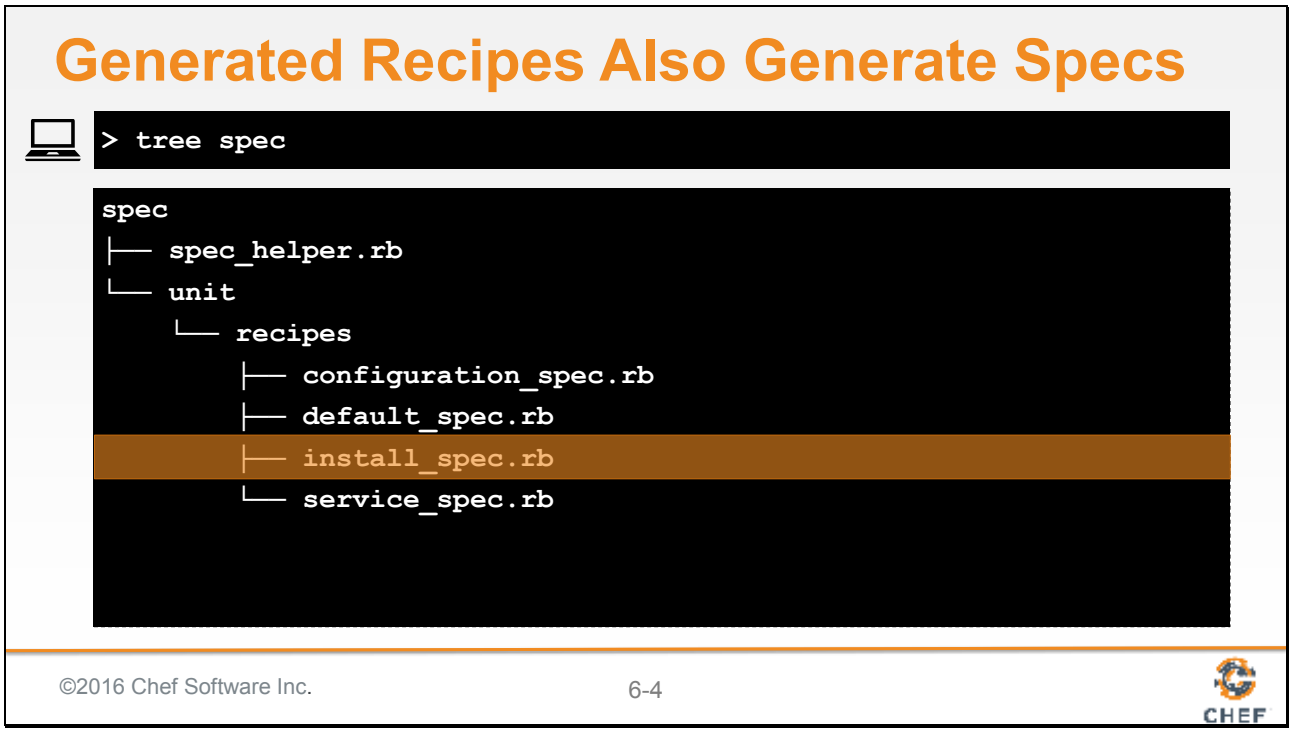
In this module you will learn how to test resources within a recipe using ChefSpec.

Slide 3



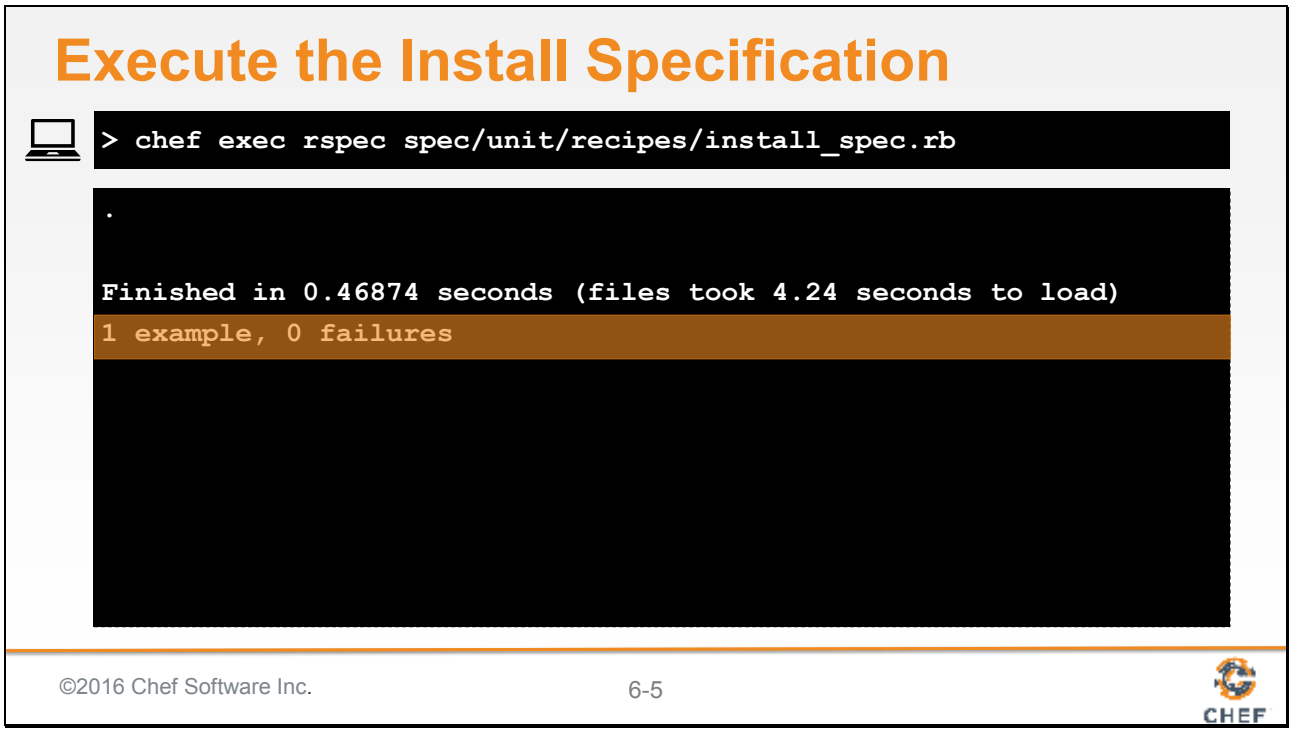
If we continued to use the mutation testing approach we would find similar problems with in the other recipes that we developed. Together let's work through defining examples for this recipe and then you will have a lab later to complete the remaining recipes.

Slide 4



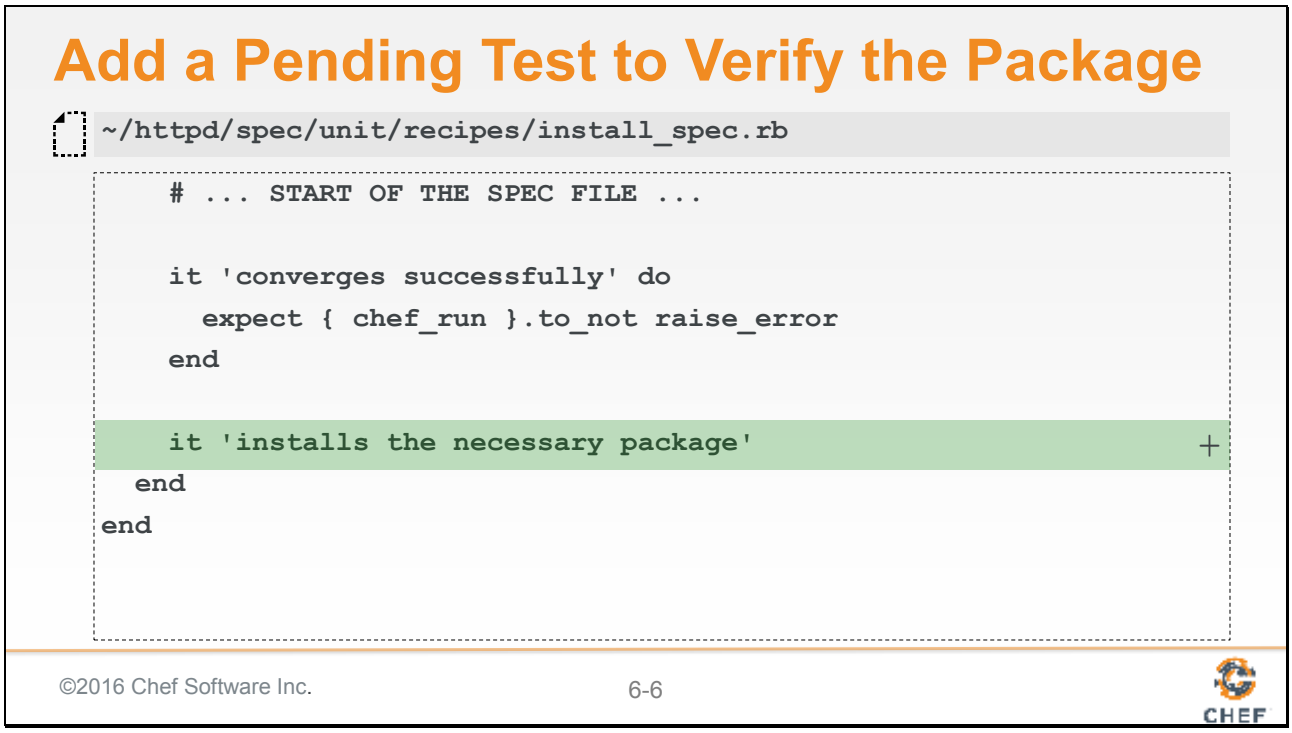
Back when we generated the recipe with the 'chef' command-line utility a matching specification file was also generated. Similar to the default recipe specification the install recipe specification contains a single example that ensures that the chef run completes without error.

Slide 5



Using 'rspec' we can verify that the one example completes successfully.

Slide 6

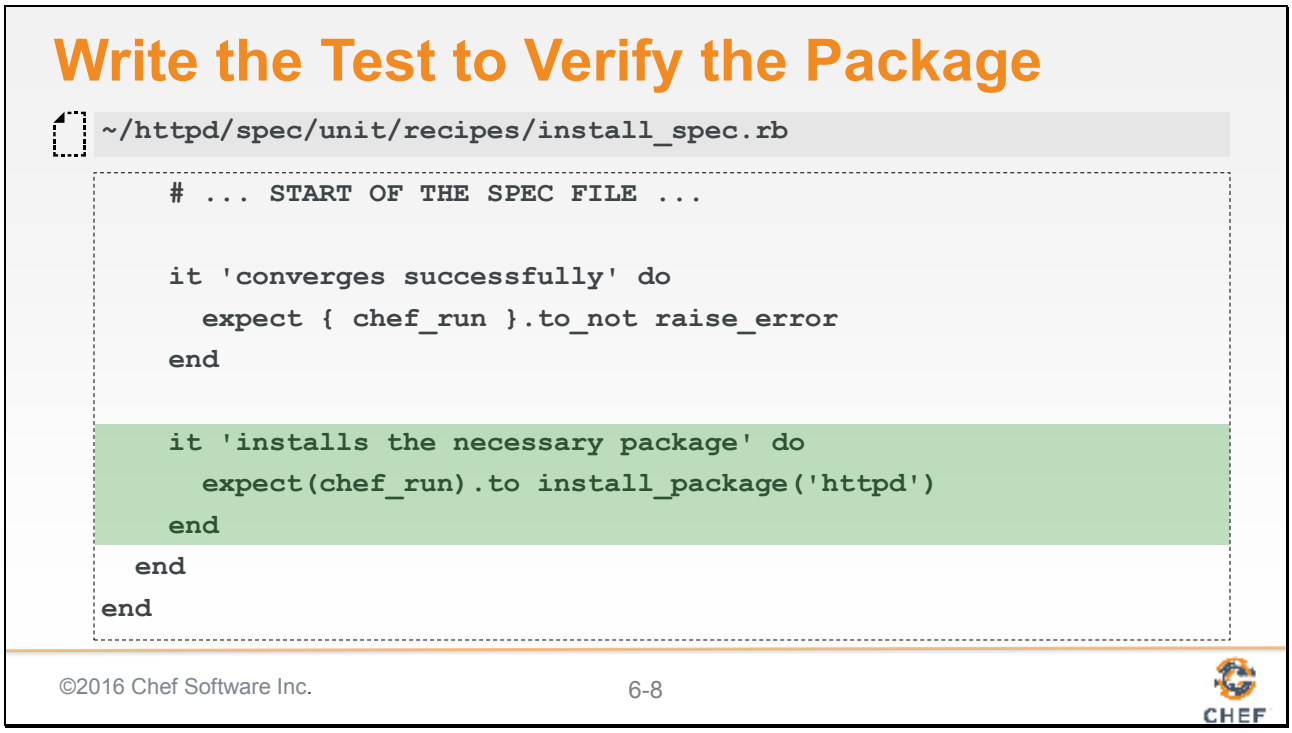


The install recipe installs the necessary the necessary software for the webserver. We can start by writing a pending example.

Slide 7

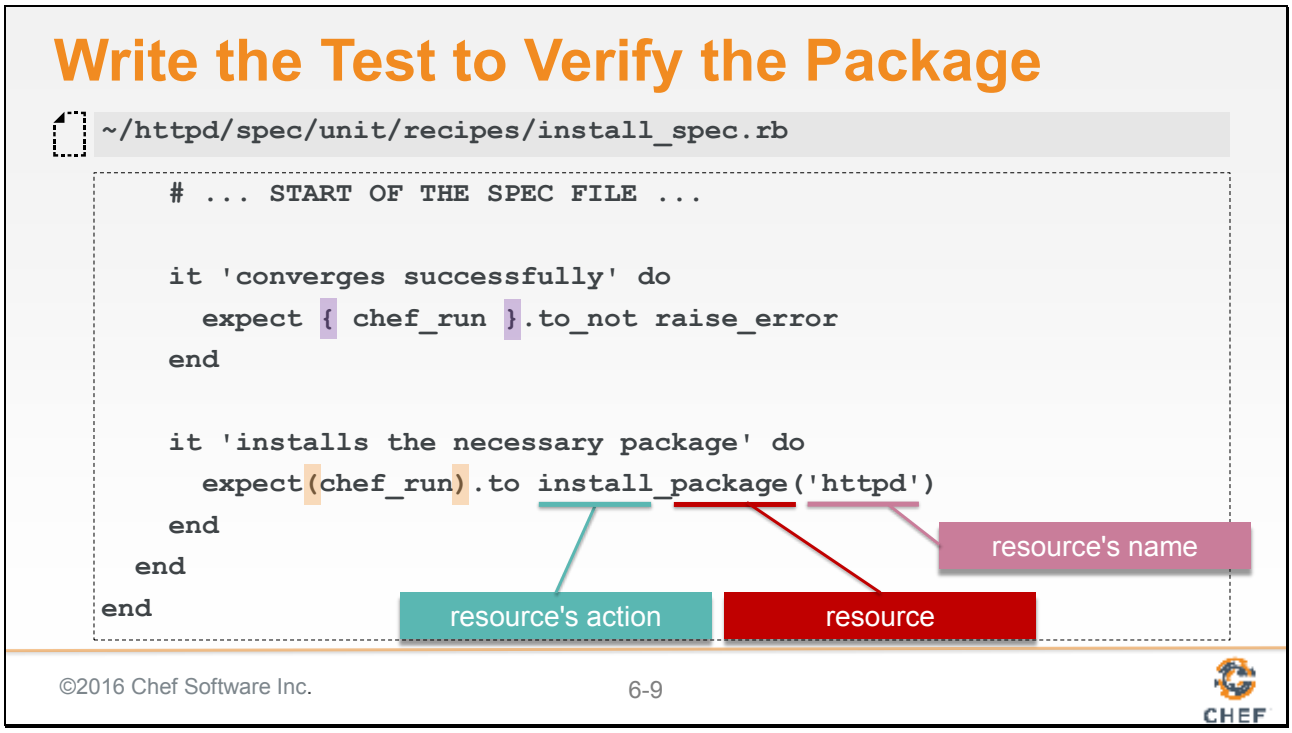
Now it is time returned to the documentation. Again, the ChefSpec documentation contains a lot of examples in the README and the examples directory. Using either of those find an example of an expectation expressing that a packaged is installed.

Slide 8



With a good example we found in the documentation we can return to the example and define the example. In our case we want to assert that the the chef run installs the package 'httpd'.

Slide 9

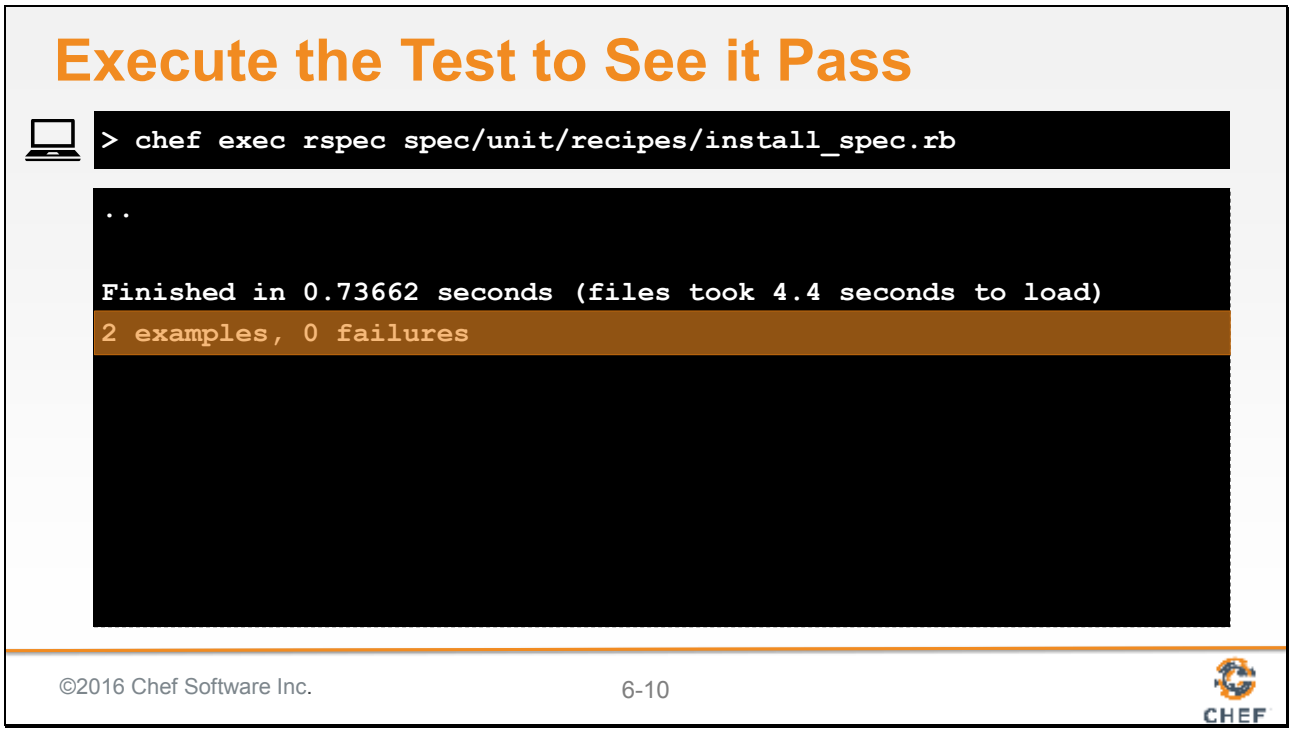


Expressing an expectation for the state of resources in the 'Resource Collection' uses a particular matcher. Express the name of the action joined together with the type of the resource and has the parameter that is the name of the resource.

The expectation defined here is slightly different than the previous example. In the first example the expect uses braces. This is Ruby's shorthand notation to represent a block. The reason in this expectation we want to use a block is that if the chef run were to raise an error we need to catch it. Catching it requires that we wrap the code we want to execute within a block.

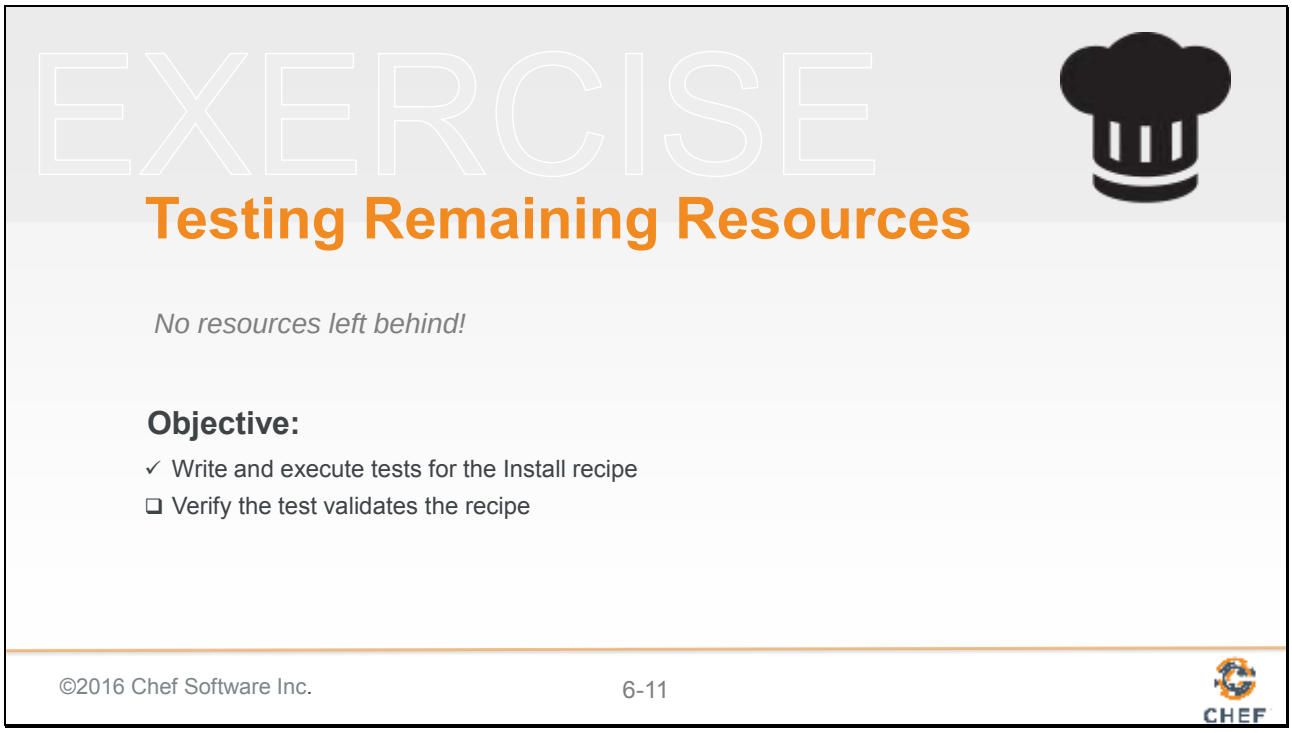
Using the parenthesis is passing the 'chef_run' helper as a parameter to the 'expect' method. In this instance we do not expect an error to take place and instead want to make assertions on the state of the chef run. If an error were to be raised the expectation would not catch it and instead of the expectation failing you would see an error message.

Slide 10



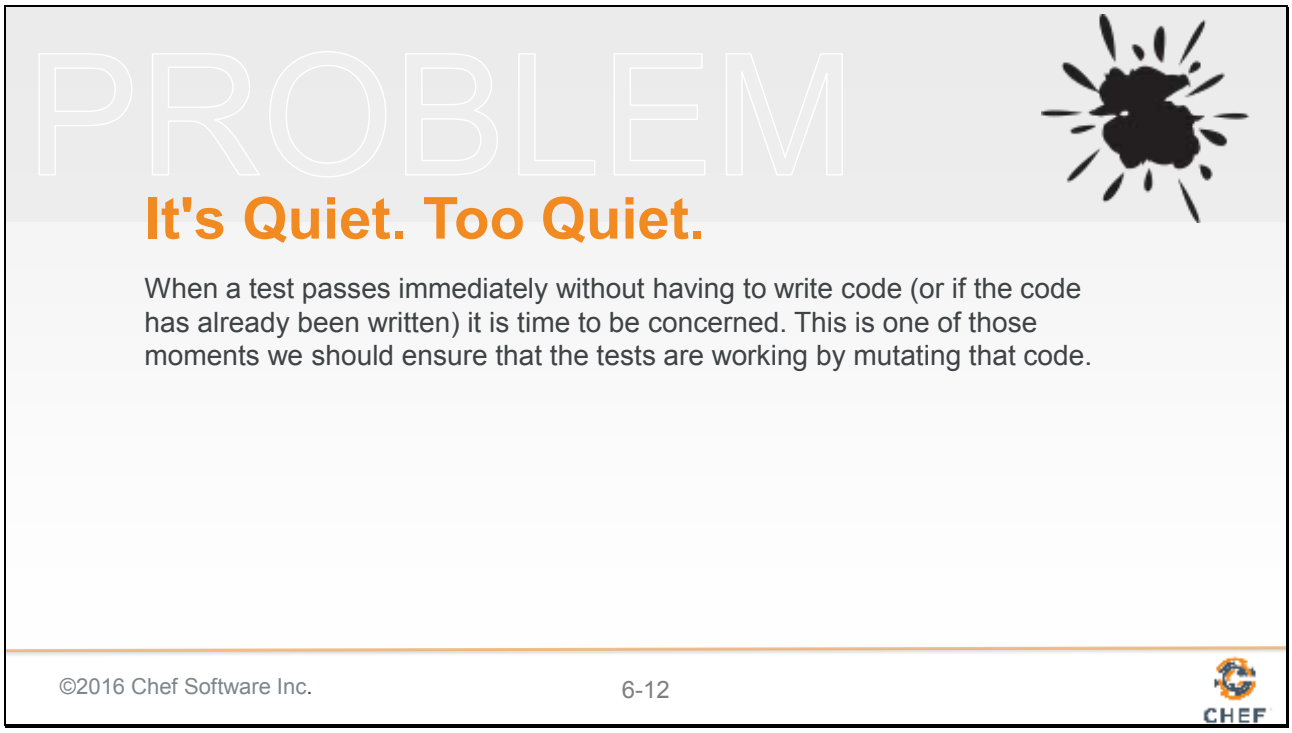
When we are done writing this expectation and execute the test suite we see that we now have 2 examples that both pass.

Slide 11



We now have an expectation that expresses the state for the install recipe. But before we declare victory it is time to verify that the expectations truly are working.

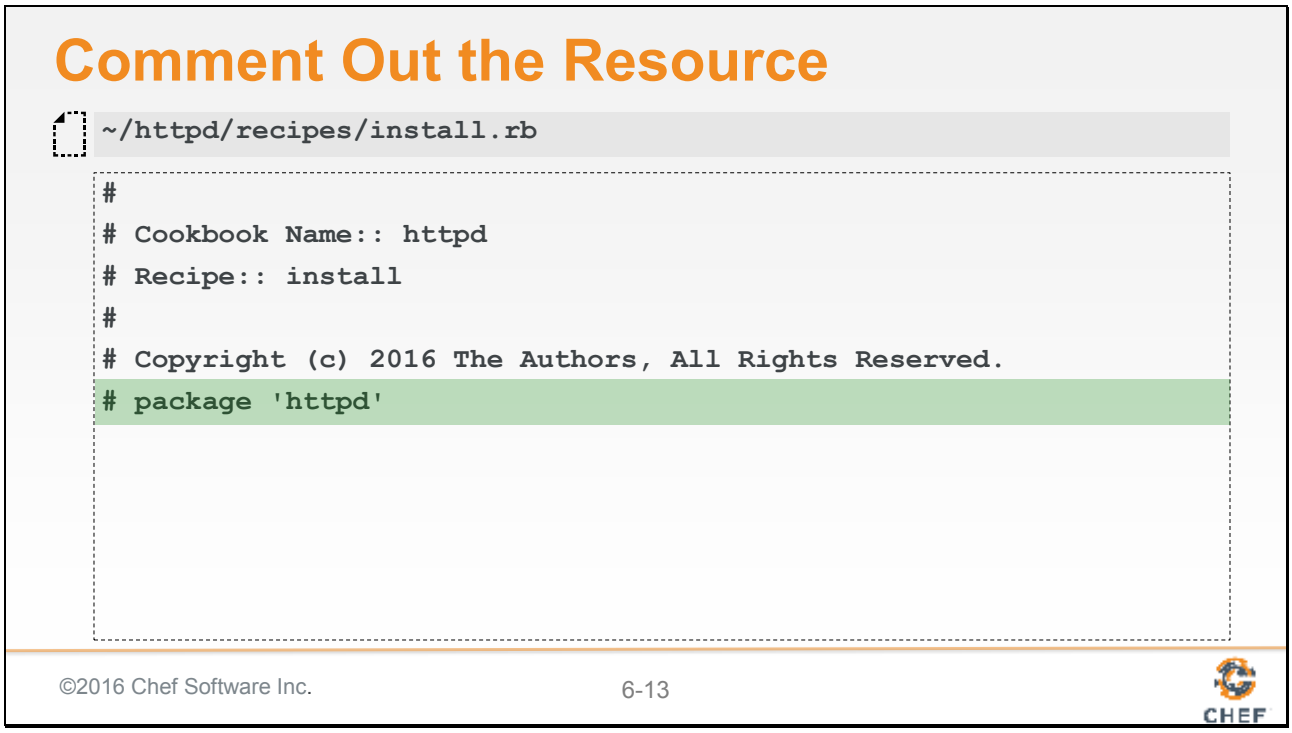
Slide 12



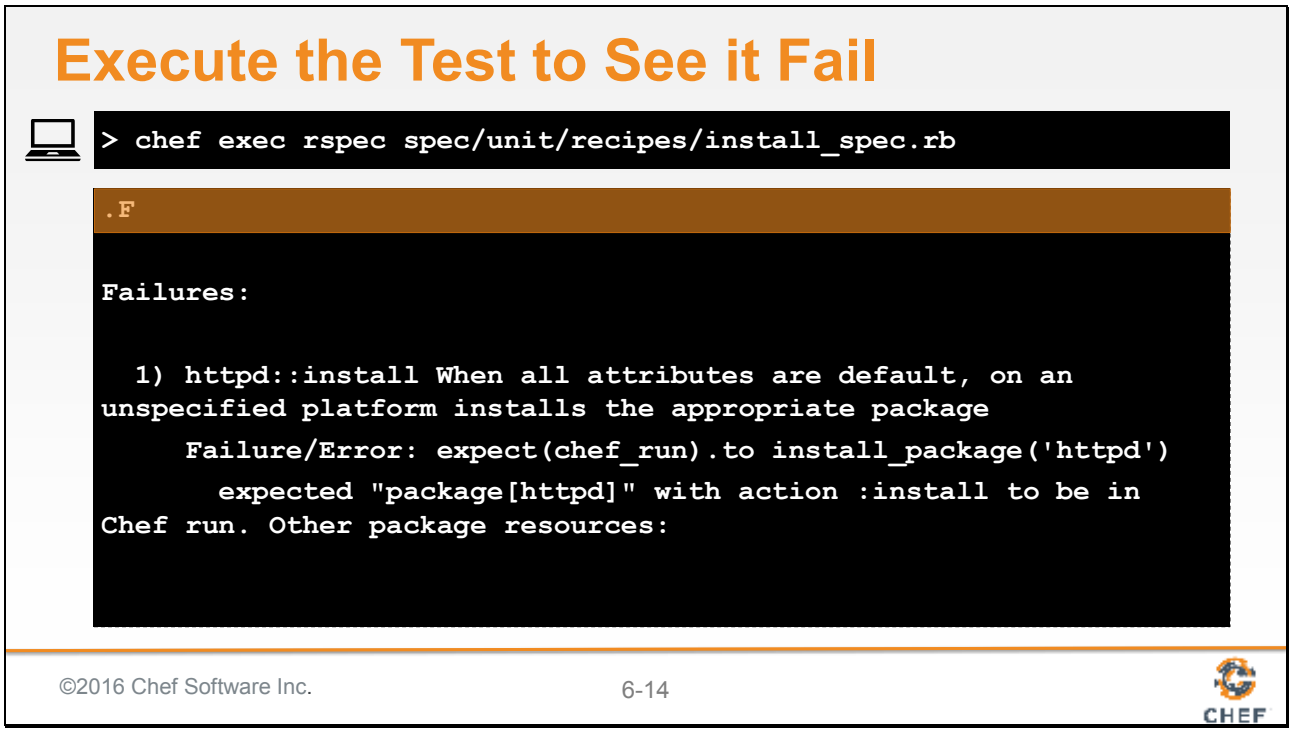
If a test passes and you have never seen it fail. How do you know it works? Without ever seeing a failure there is situation where we could be seeing a 'false positive'. This is because we did not develop this expectation with the test first. In this instance we have not done anything wrong. We simply need to ensure that the expectation we define will fail if we were to modify the code that we are testing.

To do that it is time for us to return to the recipe and modify it, mutate it, to ensure that the test fails.

Slide 13



One simple mutation is to remove the resource by commenting it out or removing it. We could also choose to rename the name of the resource.



Returning to run the tests we now see that there is an error in the execution. The change that we made to the recipe, the removal of the resource, generates this error. We can state with confidence that the expectation that we defined properly insures our expectations about the 'Resource Collection'.

Slide 15

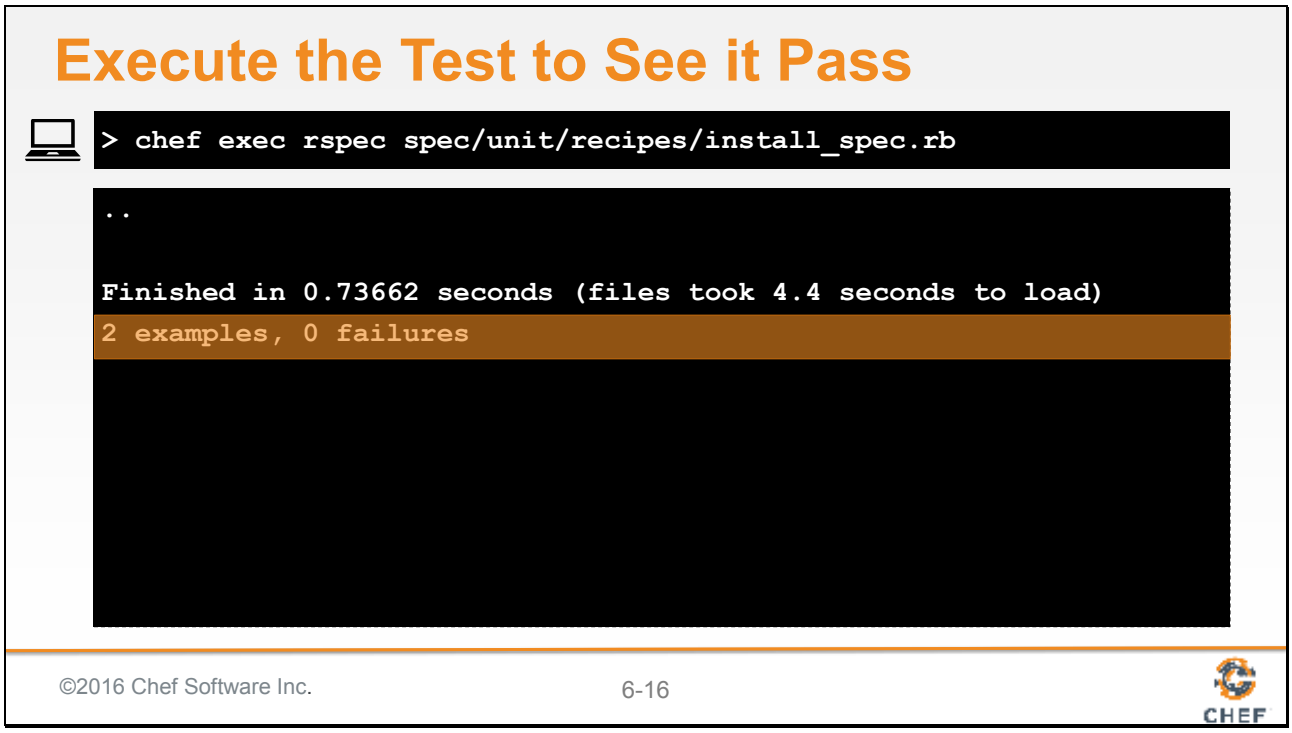
Uncomment Out the Resource

~/httpd/recipes/install.rb

```
#
# Cookbook Name:: httpd
# Recipe:: install
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
package 'httpd'
```

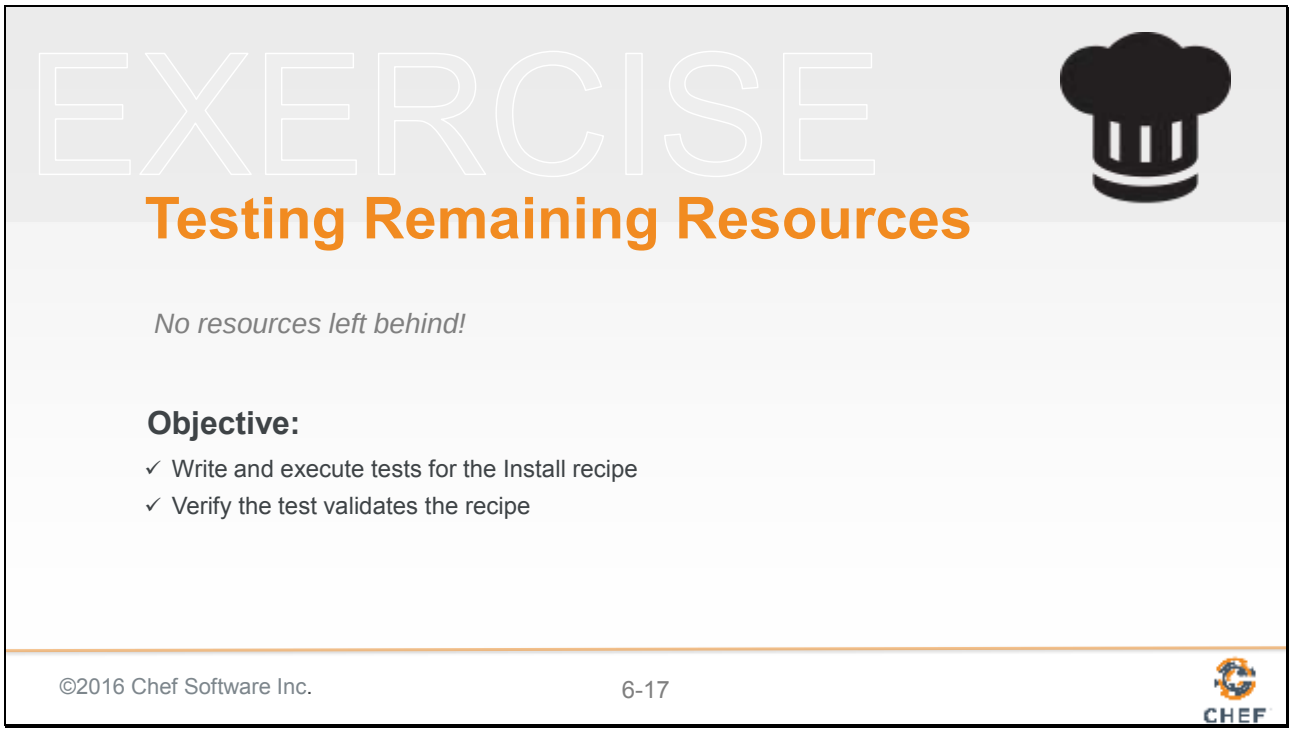

Time to restore the mutation we introduced.

Slide 16



Verify that all the examples complete successfully.

Slide 17



We walked through ensuring this recipe has the necessary expectations defined.

Slide 18

LAB

Test the Remaining Recipes

- ❑ Write a pending example
- ❑ Find the ChefSpec implementation
- ❑ Verify that the new example passes
- ❑ Mutate the recipe to generate a failure
- ❑ Restore the code in the recipe
- ❑ Verify that all examples pass

❖ **Repeat this series of steps for the configuration recipe and service recipe**

CHEF

Now it is your turn. Using the same strategy it is time to address the remaining recipes within the cookbook.

Instructor Note: Allow 15 minutes to complete this exercise

Slide 19

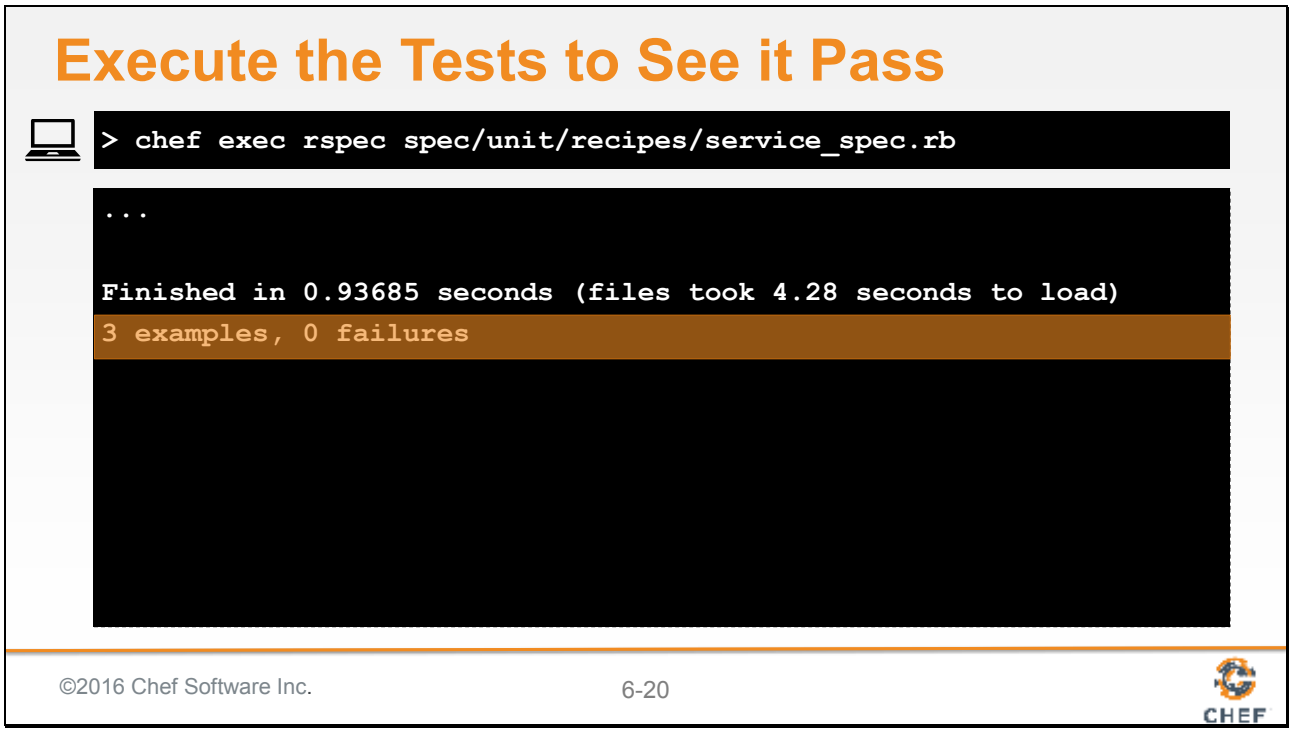
Write the Tests to Verify the Service

~/httpd/spec/unit/recipes/service_spec.rb

```
# ... START OF THE SPEC FILE ...

    it 'starts the necessary service' do
      expect(chef_run).to start_service('httpd')
    end

    it 'enables the necessary service' do
      expect(chef_run).to enable_service('httpd')
    end
  end
end
```


Let's review the final resulting specification for only the service recipe. We defined two examples. One that states the expectation that the necessary service has been started. The other states the expectation that the necessary service has been enabled.

Instructor Note: We are showing the final concluding content and not the workflow.



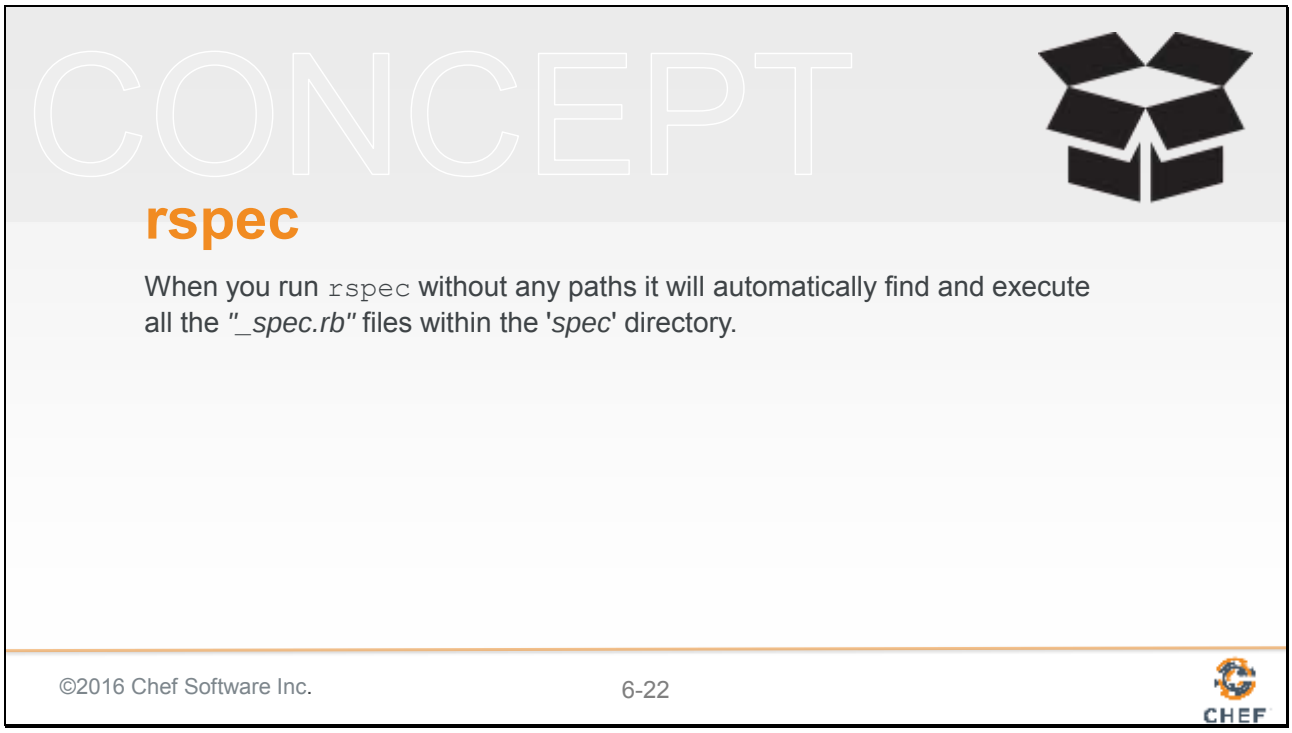
Verifying the examples we see three passing examples.

Slide 21

Congratulations. Now you have completed writing unit tests for the remaining resources across all our recipes.

Instructor Note: We did not review the configuration recipe.

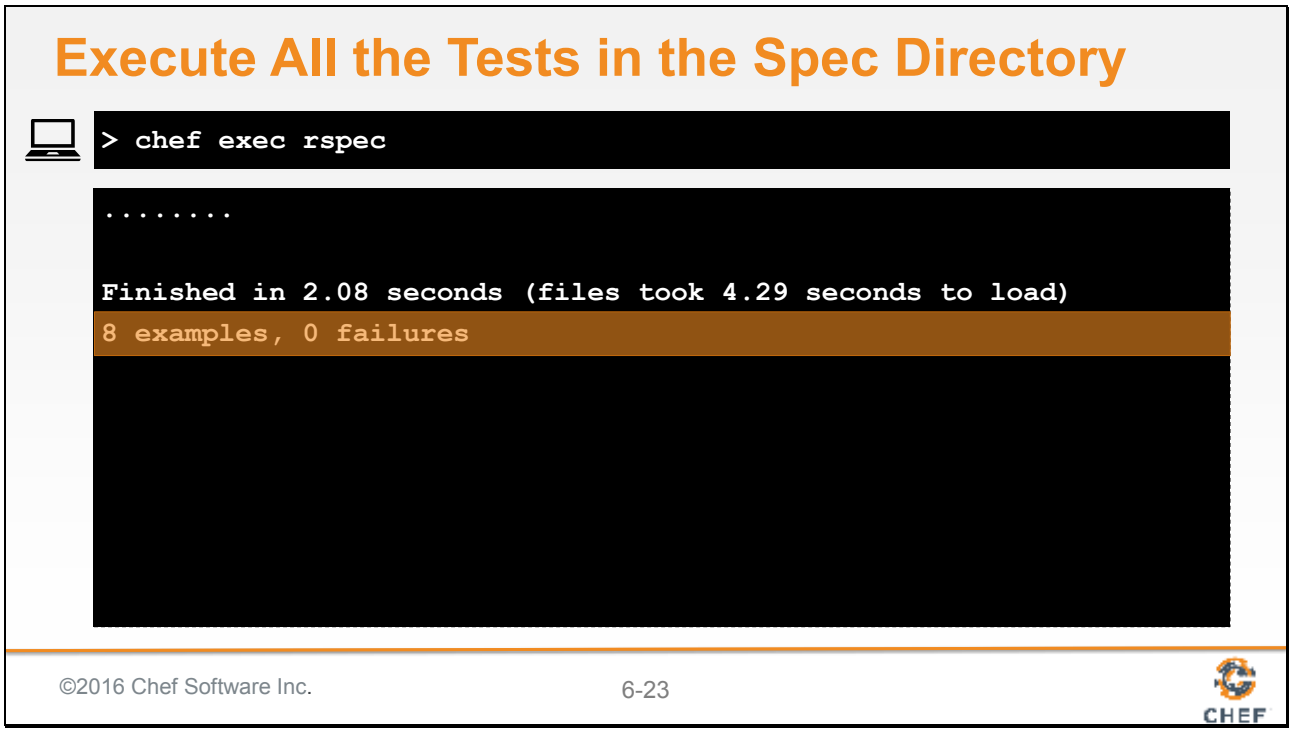
Slide 22



Running 'rspec' as we have during this and the last section has shown that we can provide a file and it will evaluate the examples within that file. Now that we have examples spread across multiple recipes it would be nice to be able to run them all at once. And actually that is how RSpec is designed to work by default. When you run 'rspec' with no paths it will automatically find all specification files defined in the 'spec' directory.

It is important to note that all specification files must end with an '_spec.rb' for them to found by RSpec.

Slide 23



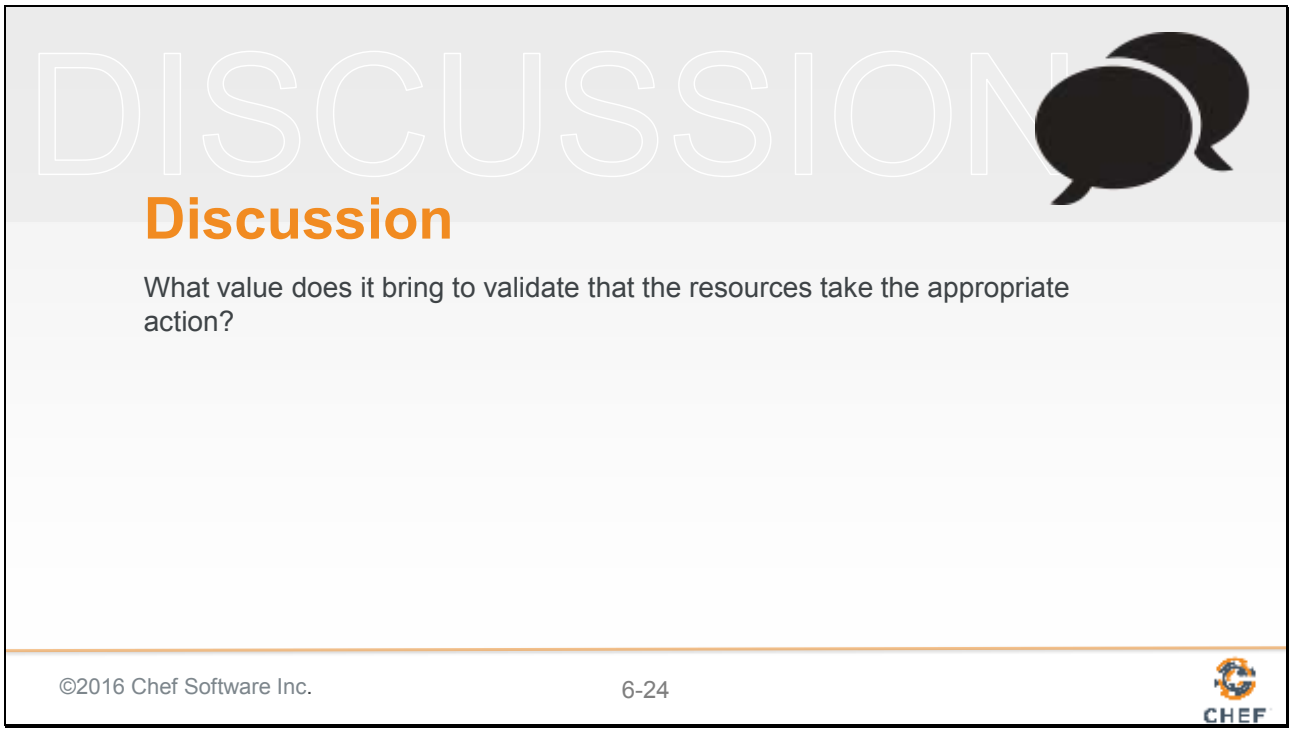
Let's verify every example across all the recipe specification files. In this output we see 'rspec' found 8 examples found all of them passing all within 4.29 seconds.

The execution time of RSpec varies based on the specifications, the version of ChefSpec, the power of the workstation, and the platform.

Let's have a discussion.

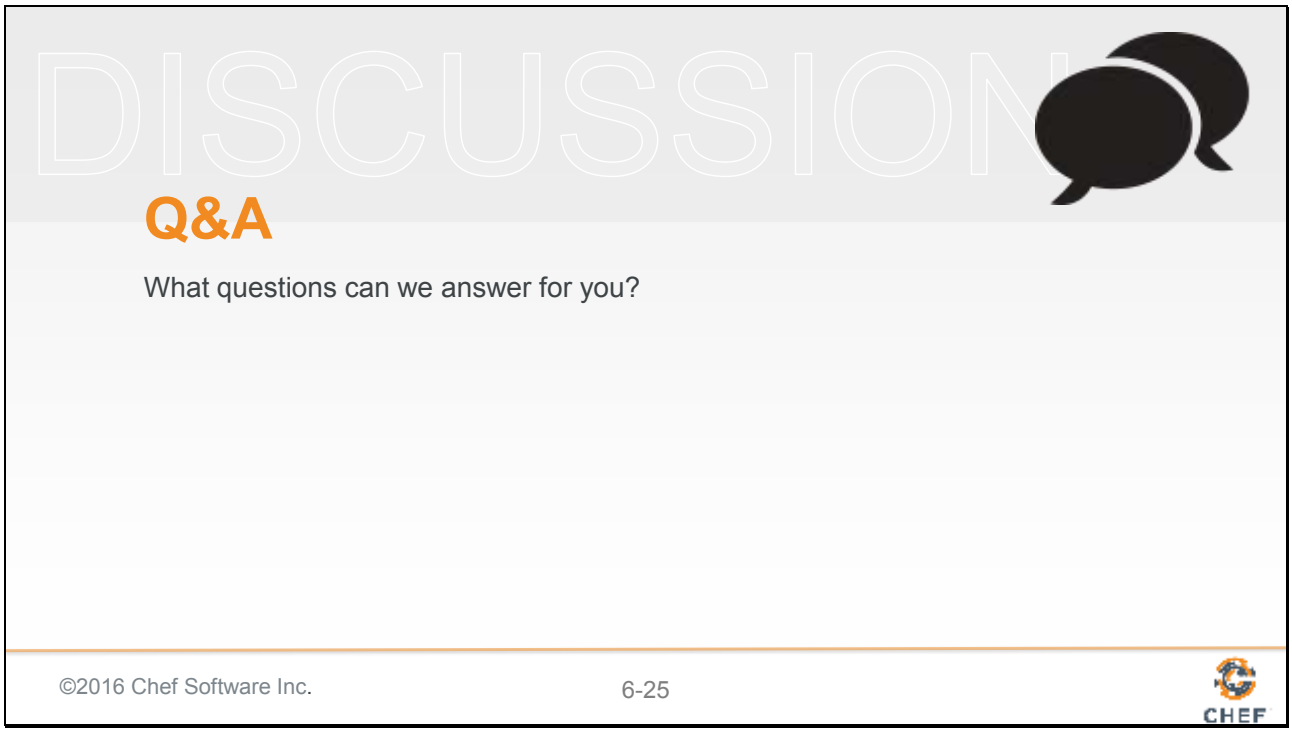
Instructor Note: This output was generated on a Amazon Web Services t1.micro running CentOS 6.7 installed with Chef DK 0.11.0.

Slide 24



Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 25



Before we complete this section, let us pause for questions.

Slide 26

Morning

Introduction
Why Write Tests? Why is that Hard?
Writing a Test First
Refactoring Cookbooks with Tests

Afternoon

Faster Feedback with Unit Testing
Testing Resources in Recipes
Refactoring to Attributes
Refactoring to Multiple Platforms

All of the resources within our recipes have expectations. Now it is time to see the value of the examples that we have defined by returning to the recipes we wrote and introduce a new requirement: using node attributes.

Slide 27

7: Testing While Refactoring to Attributes

We now have the fastest feedback open source software can buy us! And right on time because it is time to refactor the cookbook again.

Slide 2

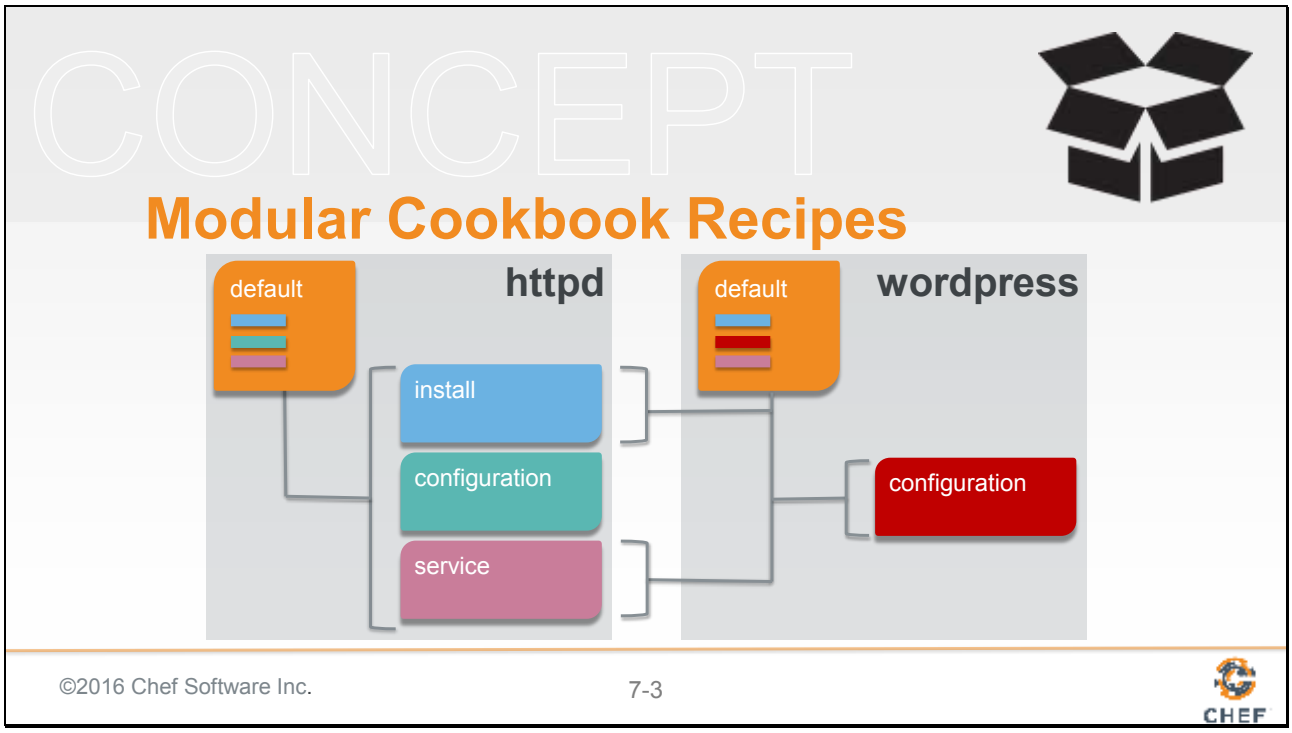
Objectives

After completing this module, you should be able to:

- Refactor resources to use attributes
- Use Pry to explore the current state of execution
- Make changes to your recipes with confidence

In this module you will learn how to refactor a cookbook to use node attributes, employ pry to set up break points in your code, and make changes with confidence.

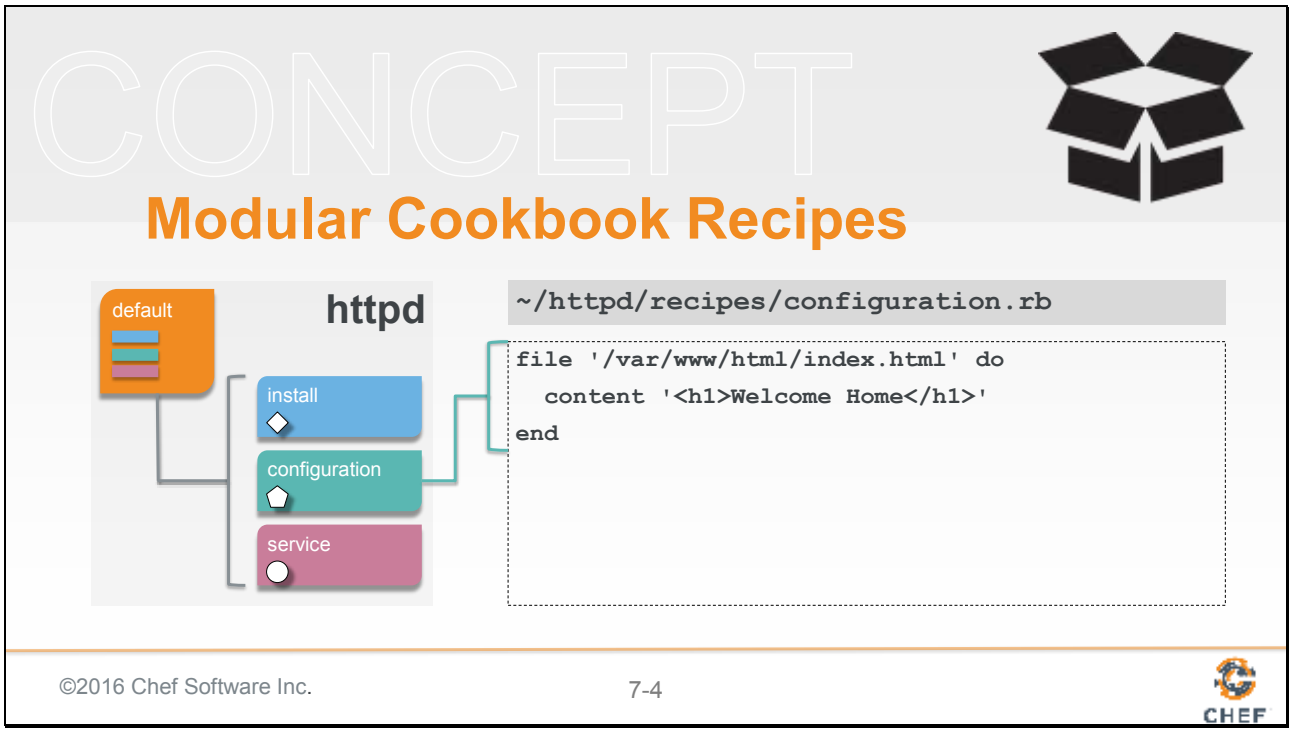
Slide 3



When we initially set out to create a cookbook that was more modular we broke the concerns of the webserver into three different recipes. This would allow an opportunity for cookbook authors within our organization re-use components of the cookbook by including only the recipes that they want.

Sometimes you do not want to re-define an entire new recipe and simply want to provide a different name or version for the package; a single file path for the configuration file.

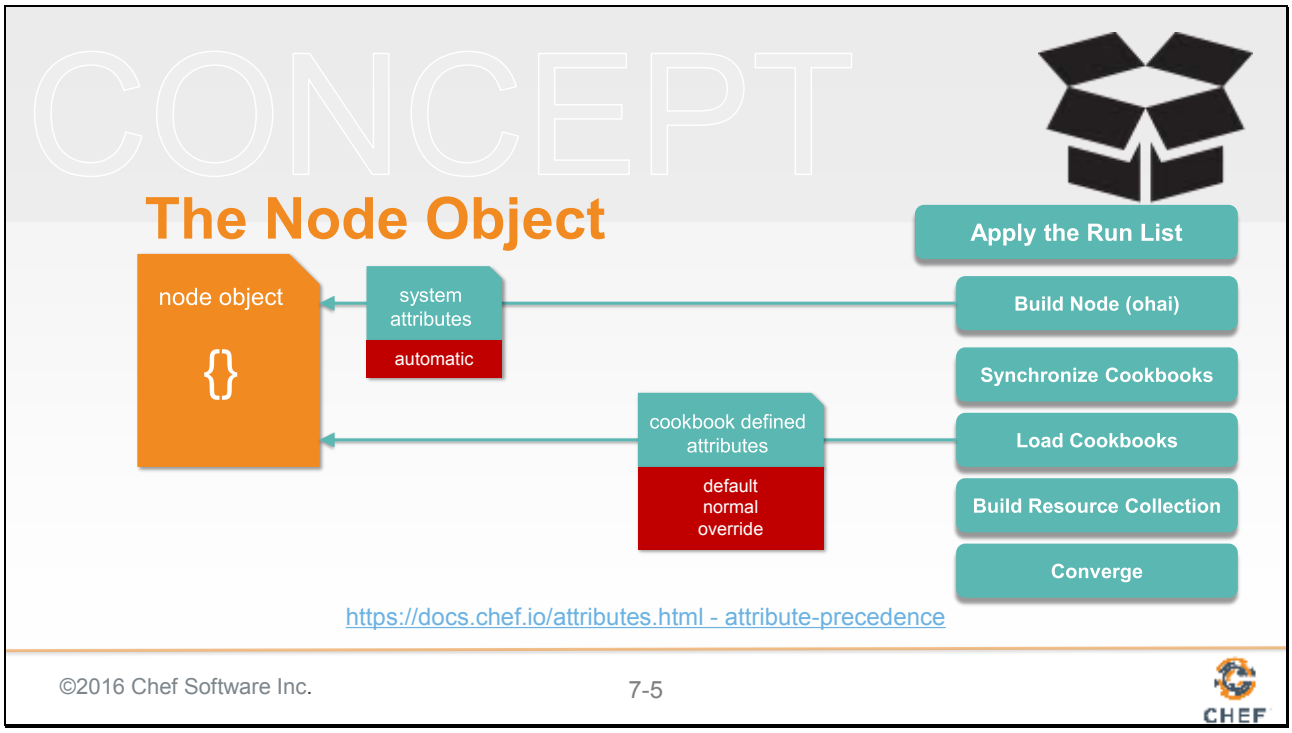
Slide 4



Within each recipe we defined the resources necessary to bring the webserver into the desired state. When we expressed these resources we did so with values that worked for this platform and version of the Operating System (OS).

The configuration recipe defined a file resource with a path to the location for the default HTML page. This path is hard-coded for this particular platform. If we had a situation where another cookbook or environment or role wanted to use this recipe but provide a custom value we could not do that unless we talk about making the file path a node attribute.

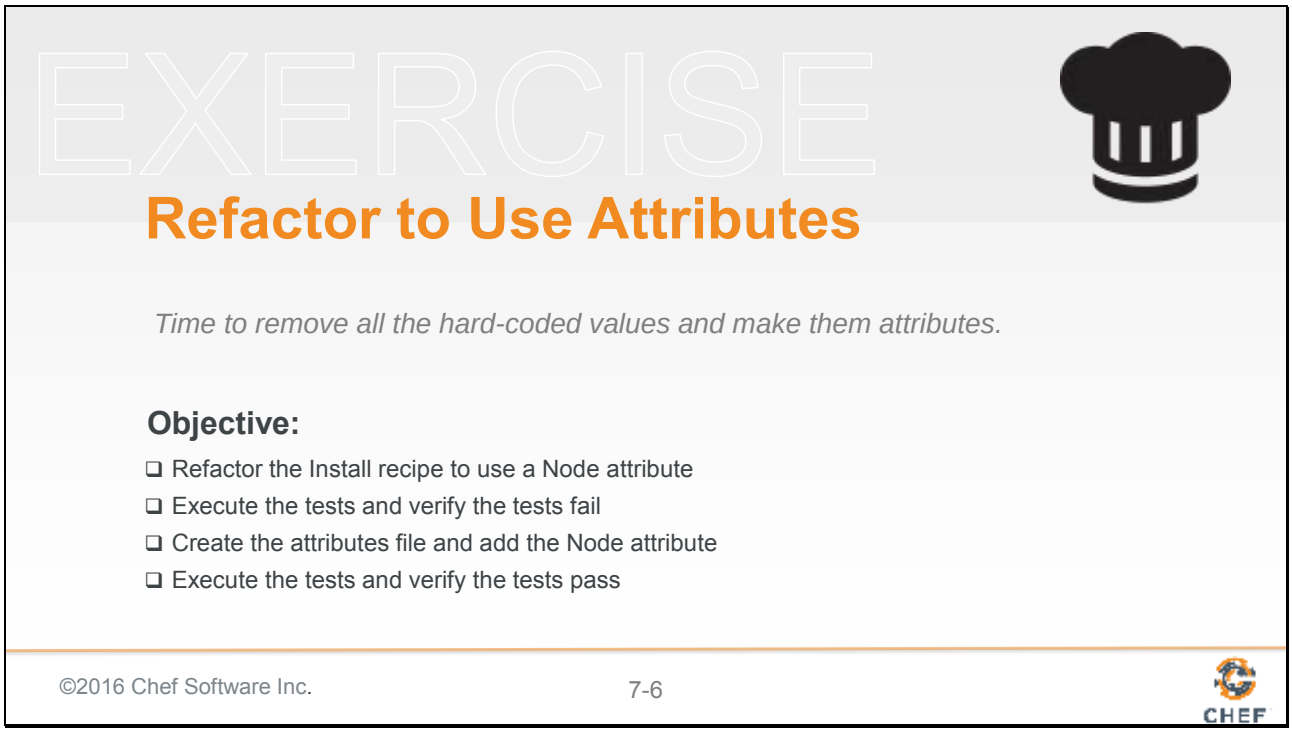
Slide 5



Cookbooks can define node attributes which are added to the node object after the initial discovery is done by Ohai. Ohai attributes are considered automatic and cannot be overwritten. However, the attributes defined in a cookbook can come in variety of levels. This allows for cookbooks to define a base value which another cookbook can replace when needed.

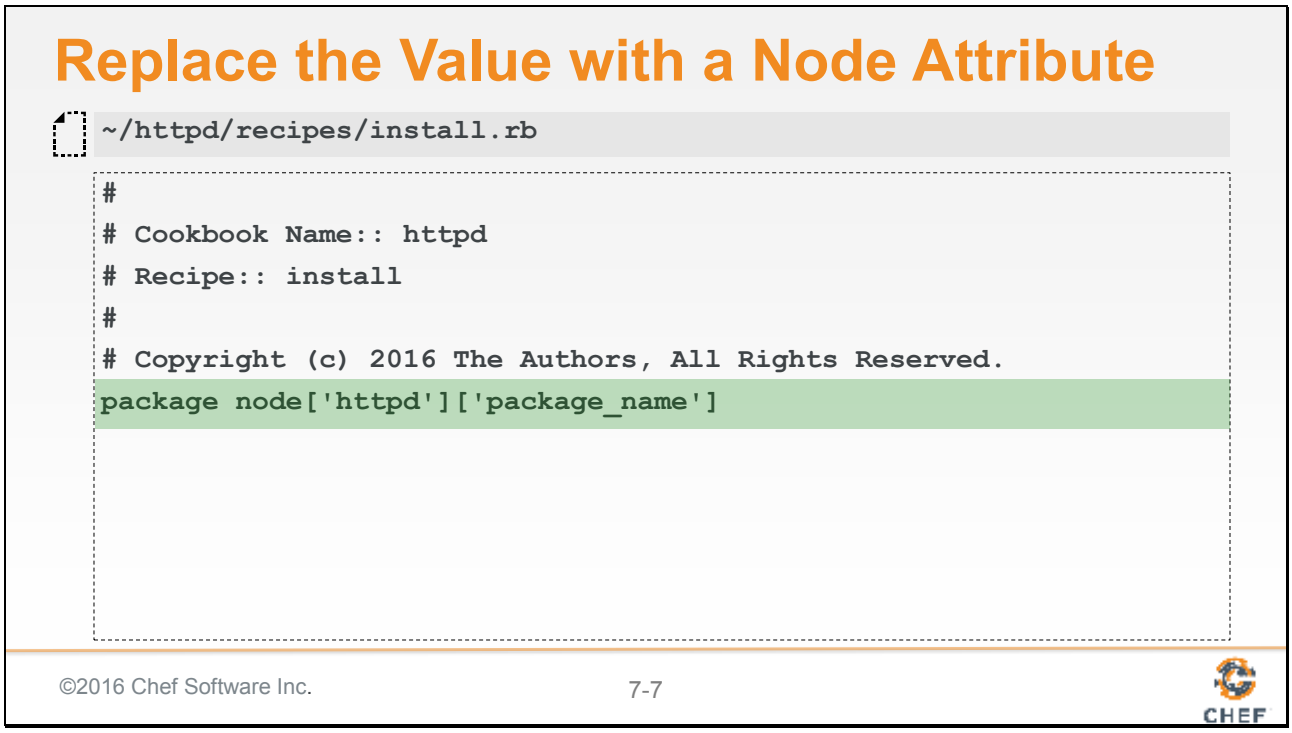
That is the kind of flexibility that we want to implement in our cookbook.

Slide 6



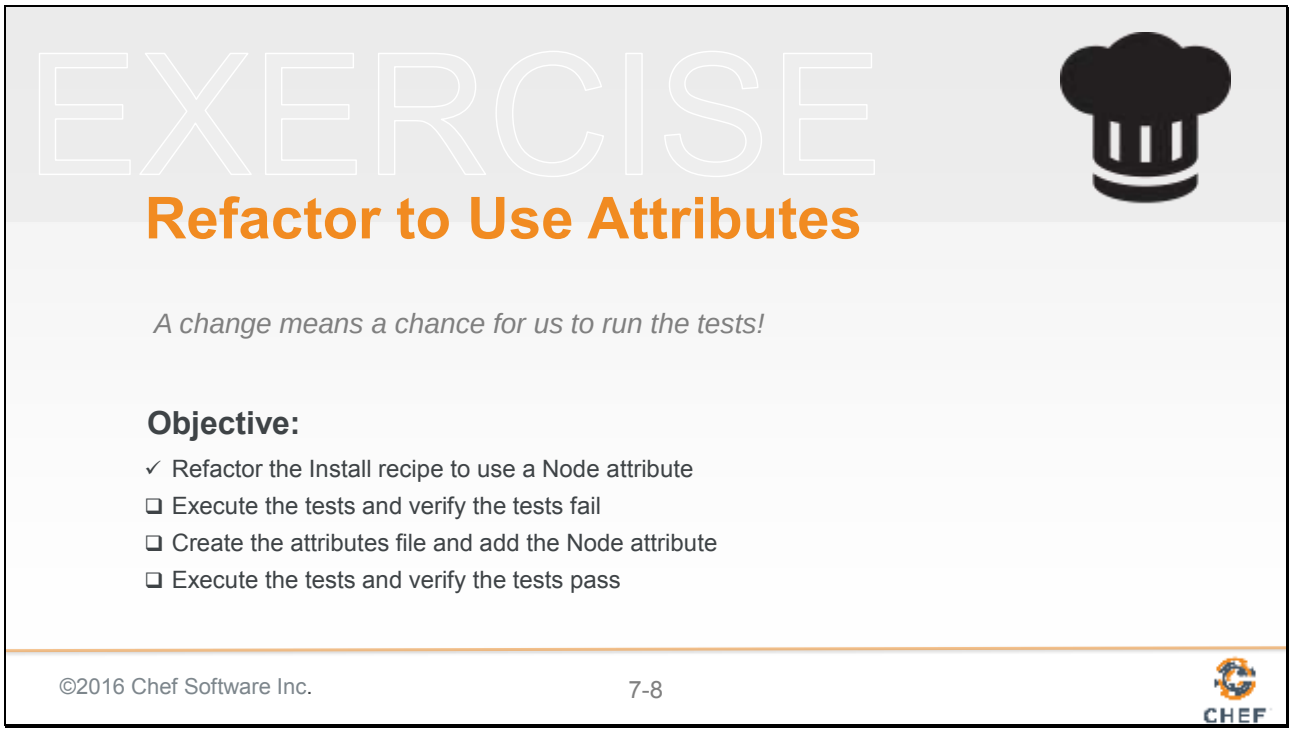
Together we will walk through refactoring the install recipe continuing to use our tests to prove that we have not caused a regression in recipes.

Slide 7

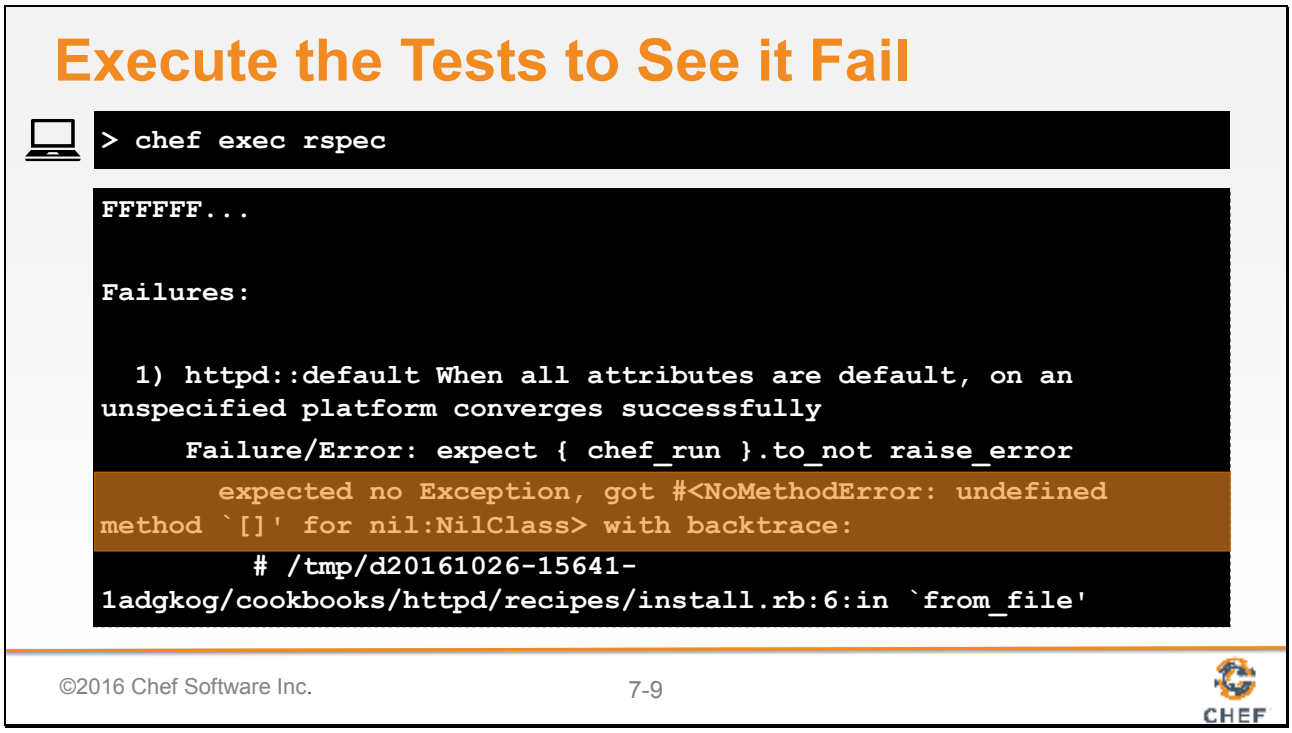


Because we have expectations in place we can start with a change to the install recipe. Here we are replacing the package name with a node attribute that we have yet to define in the attributes file.

Slide 8

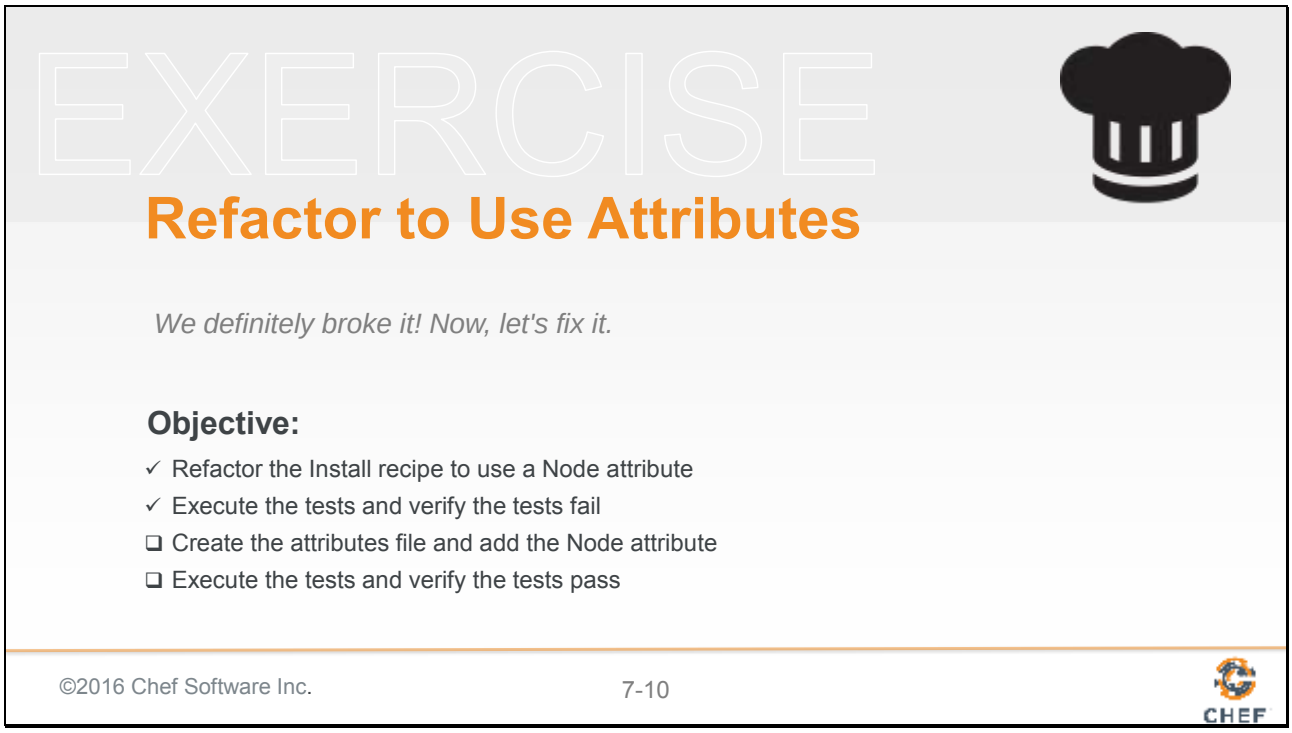


We made a change. Before we define the node attribute we should run the tests.



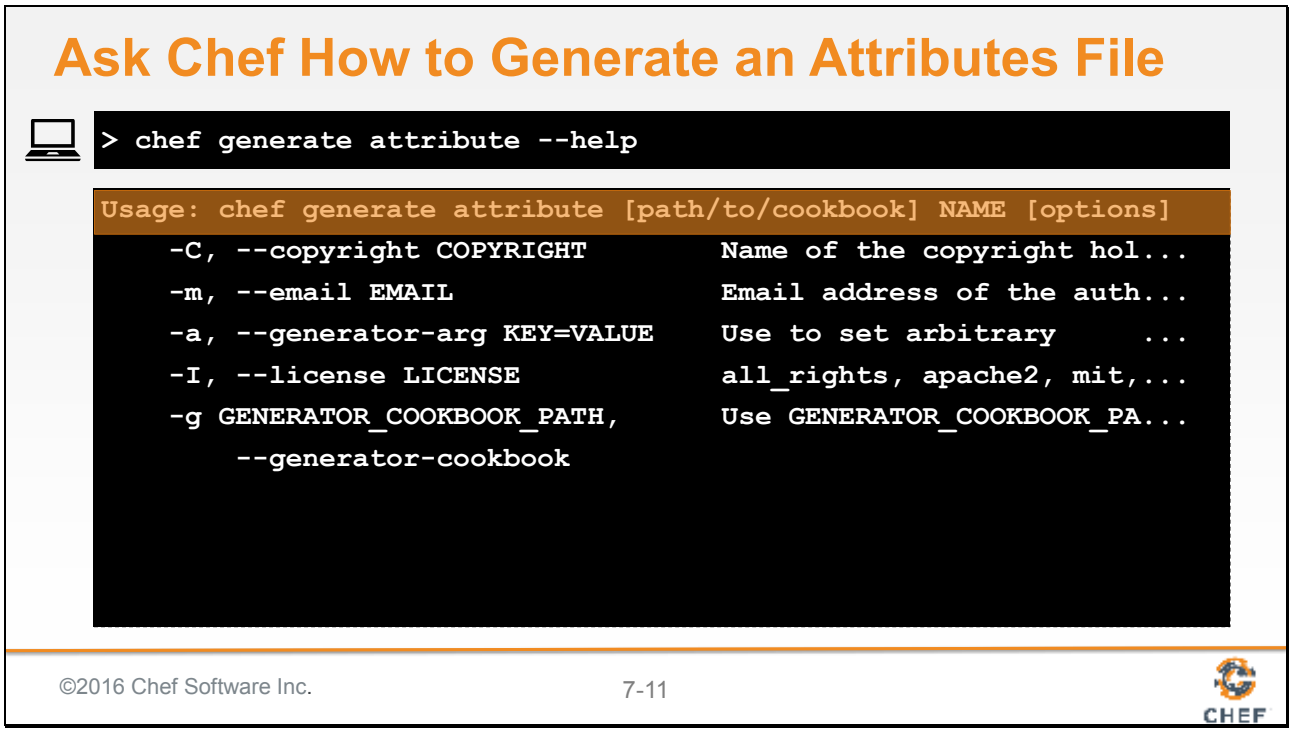
When executing 'rspec' against all the examples that we have defined we see a large number of failures. The failure summary will show us that the chef run failed with an error. This error is informing us that we attempted to retrieve an attribute from the node object that does not exist. All of the failures should be the same.

Slide 10



Now it is time to create the attributes file and define the necessary attribute.

Slide 11



The 'chef' tool is able to generate attributes. All it requires is the name of the file when you are inside the cookbook. We are currently inside the cookbook directory so now we need to give it a name.

Slide 12

Use Chef to Generate a Default Attributes File

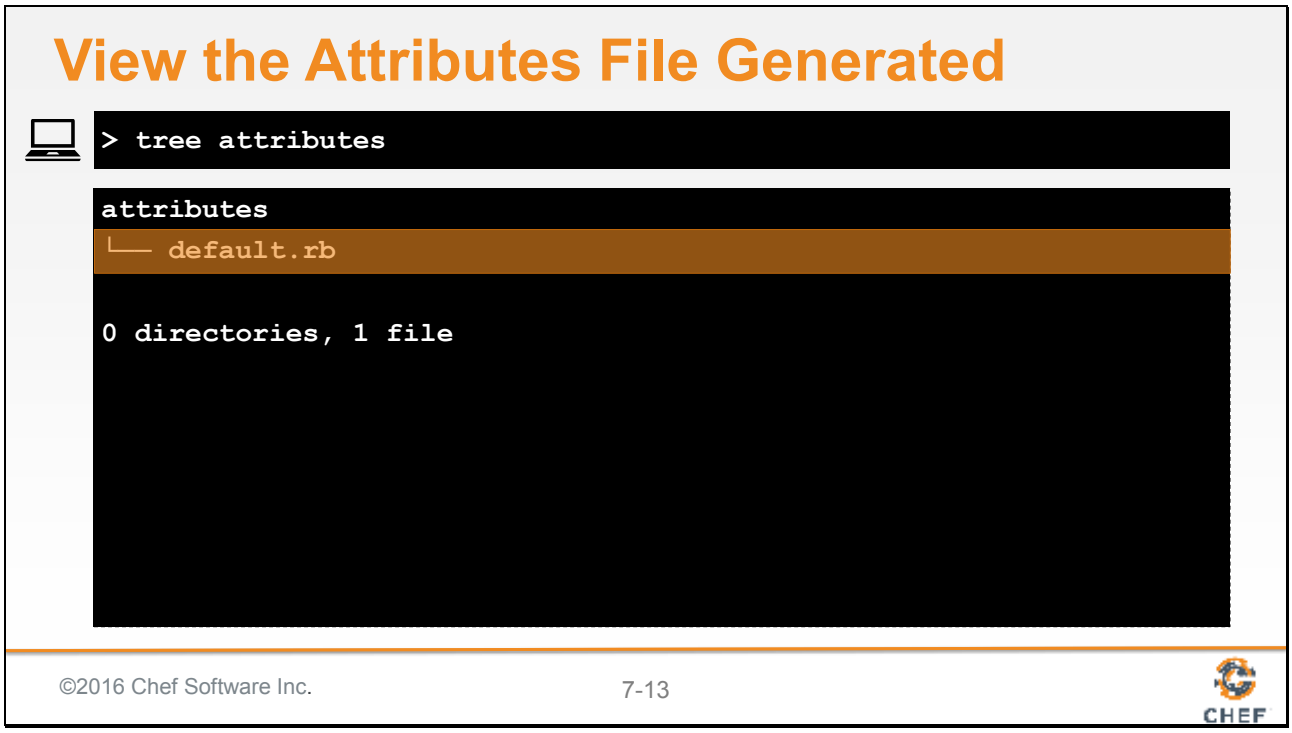
```
> chef generate attribute default
```

```
Compiling Cookbooks...
Recipe: code_generator::attribute
  * directory[/home/chef/httpd/attributes] action create
    - create new directory /home/chef/httpd/attributes
  * template[/home/chef/httpd/attributes/default.rb] action create
    - create new file /home/chef/httpd/attributes/default.rb
    - update content in file
/home/chef/httpd/attributes/default.rb from none to e3b0c4
    (diff output suppressed by config)
```


CHEF

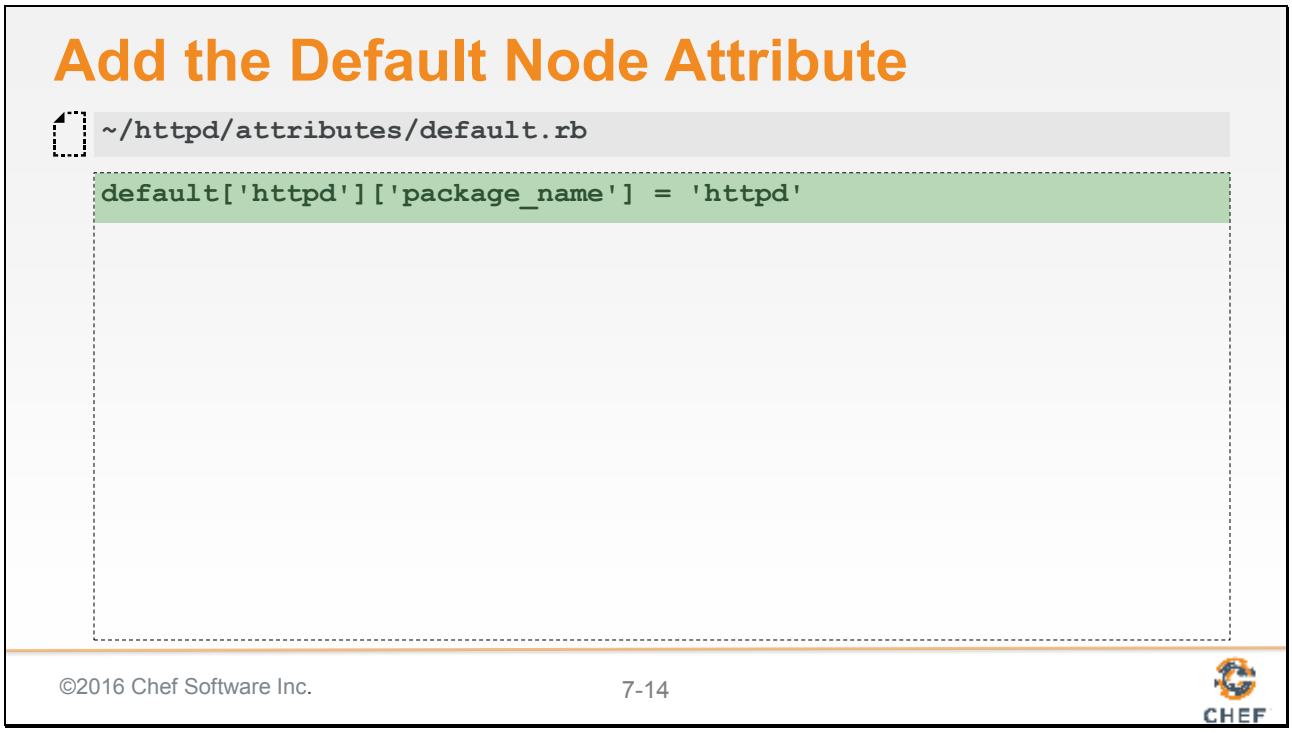
The standard name for the attributes file is 'default'. This command will generate an attributes file named 'default.rb' in the attributes directory.

Slide 13



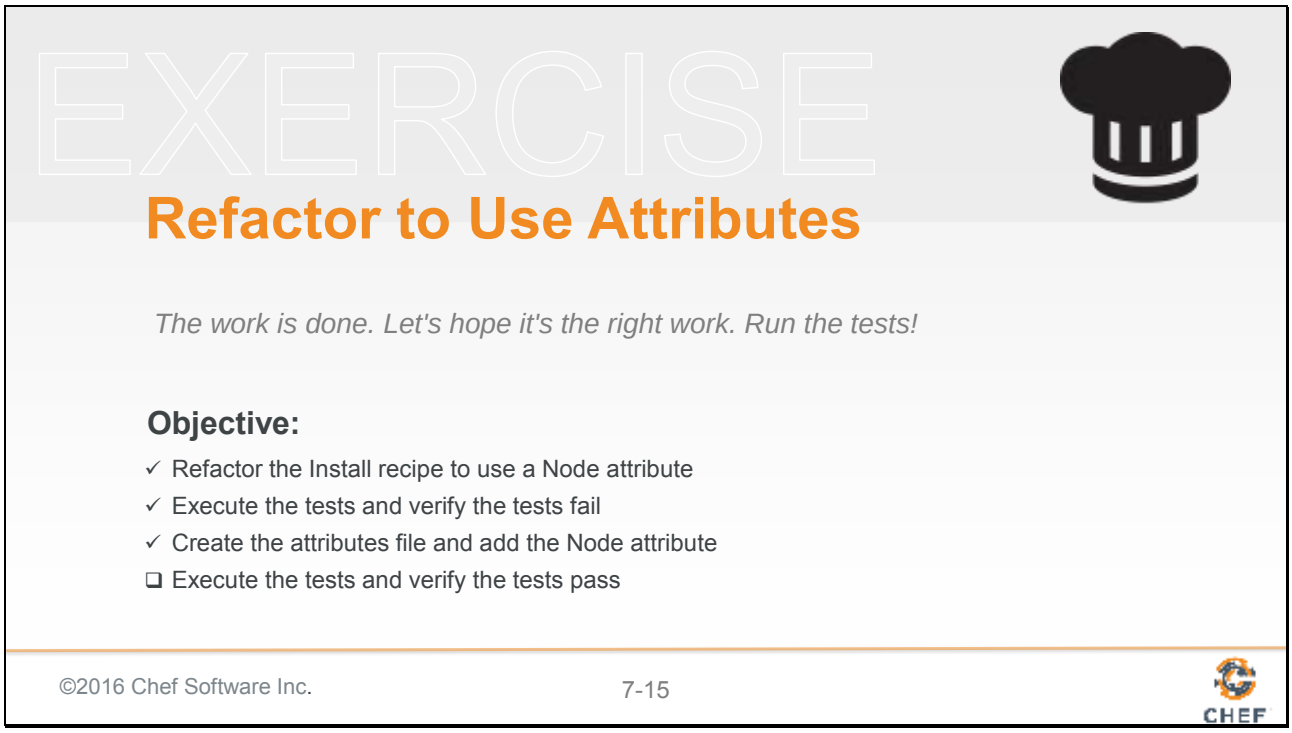
We can verify that by looking in the attributes directory to see the file has been generated.

Slide 14



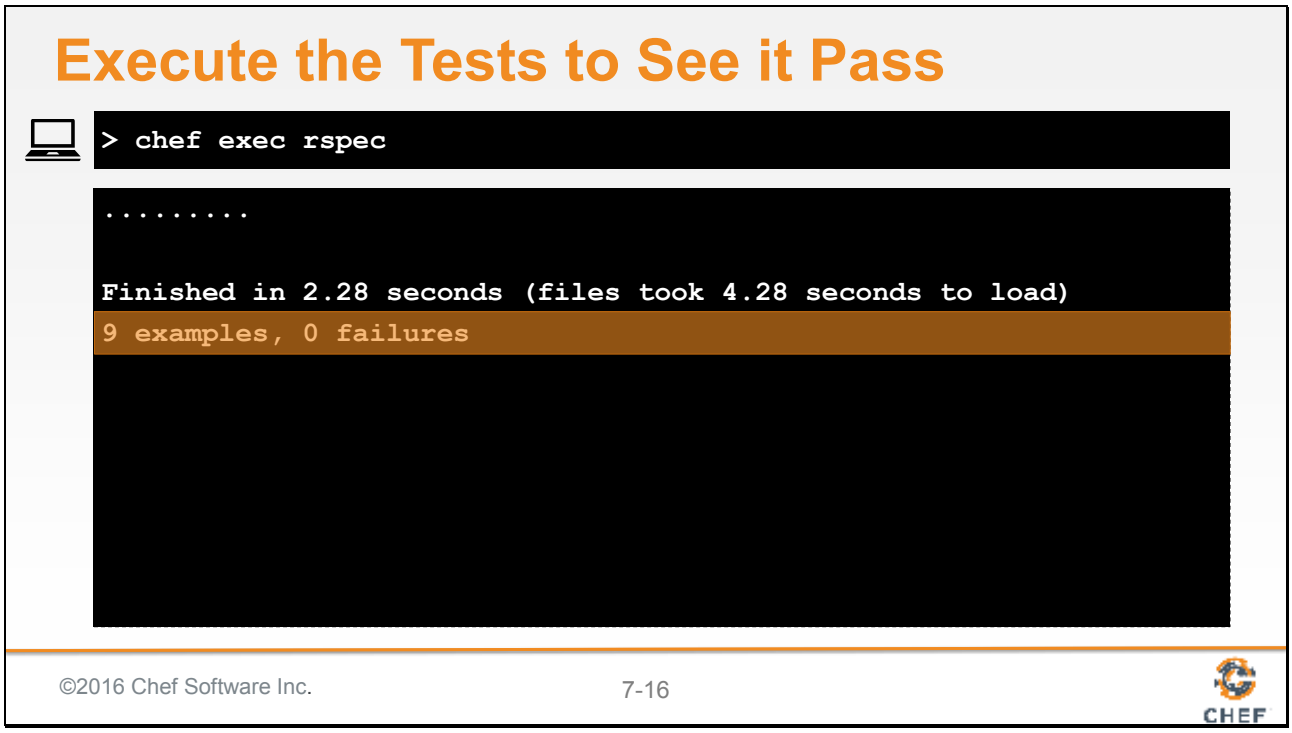
Now it is time to edit the attributes file and define the node attribute. Here we are defining the node attribute at the default level. Setting it to default will allow other cookbooks to override it if necessary.

Slide 15



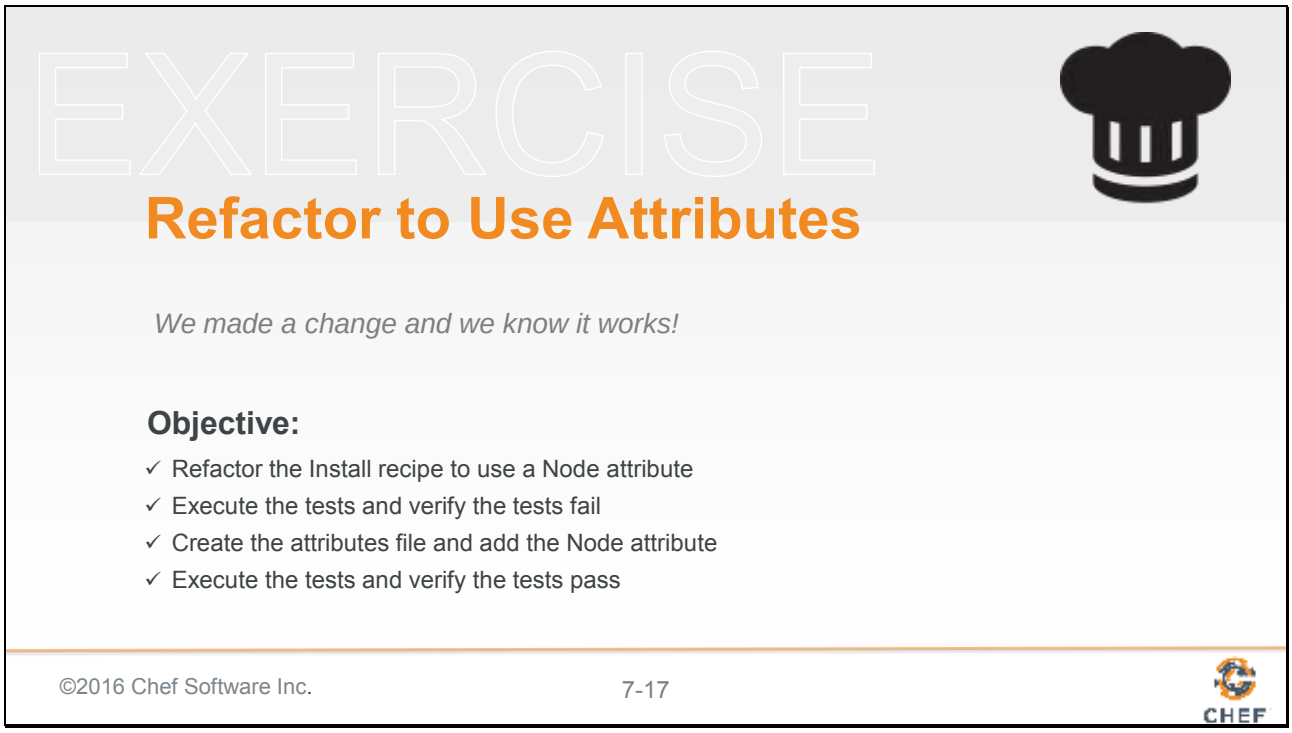
This change should fix all the examples that we broke when we used the node attribute without having defined it.

Slide 16



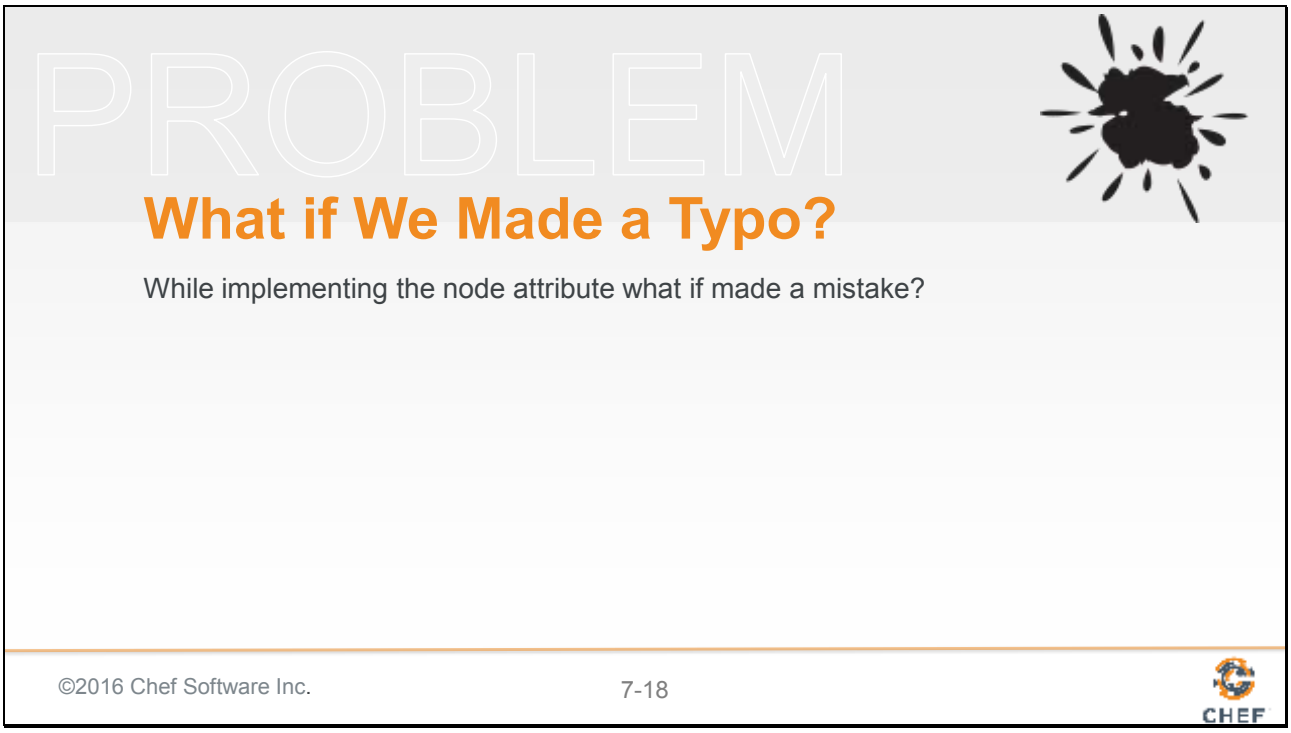
The results here show all the examples pass.

Slide 17



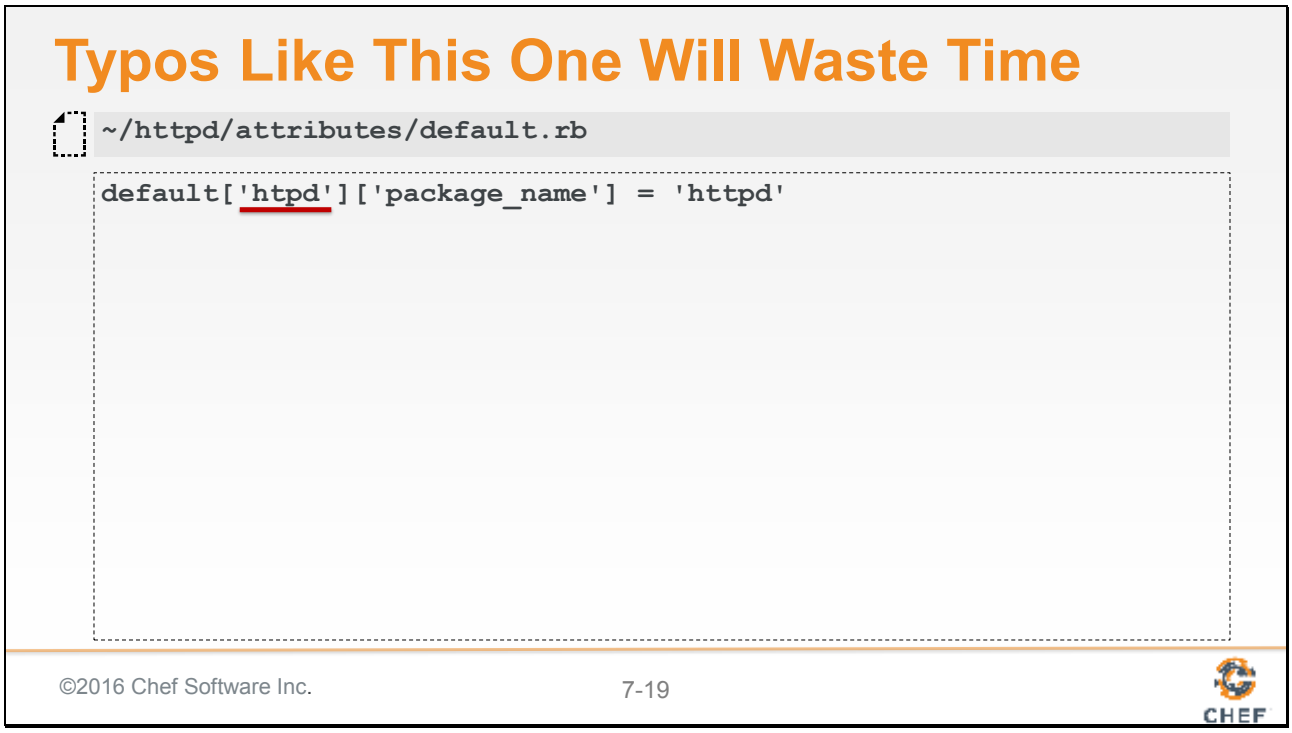
With all the expectations having been met we can confidently say that the cookbook has been refactored successfully.

Slide 18



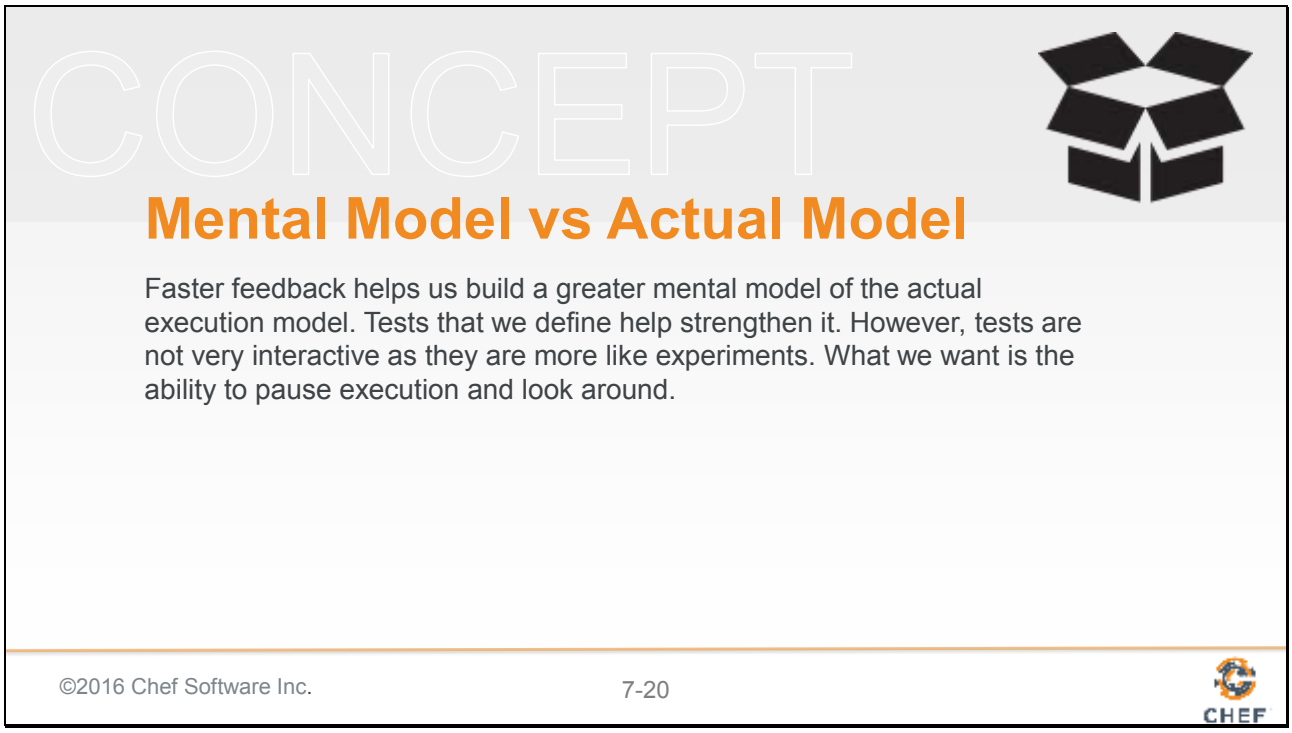
In the process of implementing the use of the node attribute in the recipe or in the attributes file we could have made a mistake. We proved that the examples would have caught the error.

What if an error occurred and we were unable to find it? Occasionally you will implement a change wrong and then find yourself staring at the failing expectations wondering what is wrong.



This is a simple typo that the examples would catch but when it comes time to find and fix the issue, our eyes may not immediately catch it. We may think the error lies somewhere in the recipe. If we cannot find it we keep running the tests and wondering what is going wrong.

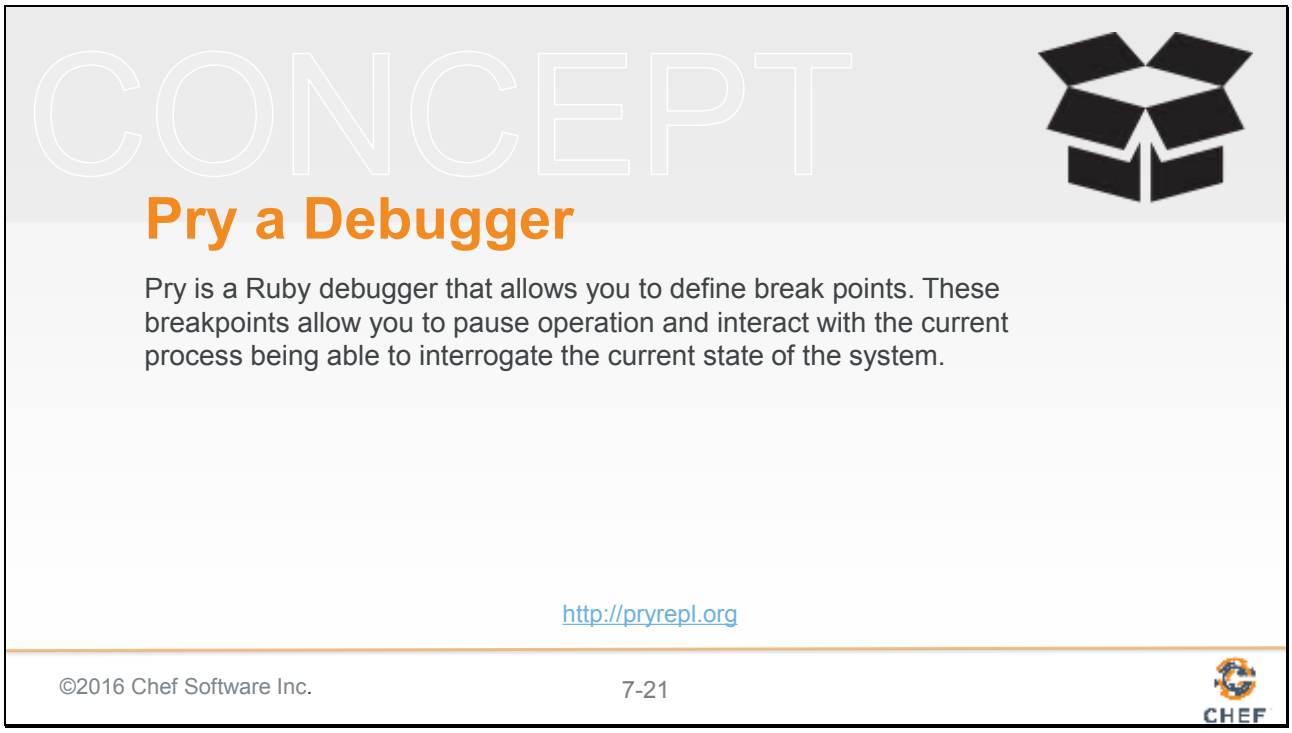
Slide 20



This is a situation where our mental model of the state of things is different than the actual model of execution. The benefit of tests is that it allows us to express the expectations about the model of how the execution should run. Testing is like a experiment: setup; execute; verify.

That feedback is not very interactive. There are moments where you want to be to stop the execution at a particular point and ask some questions.

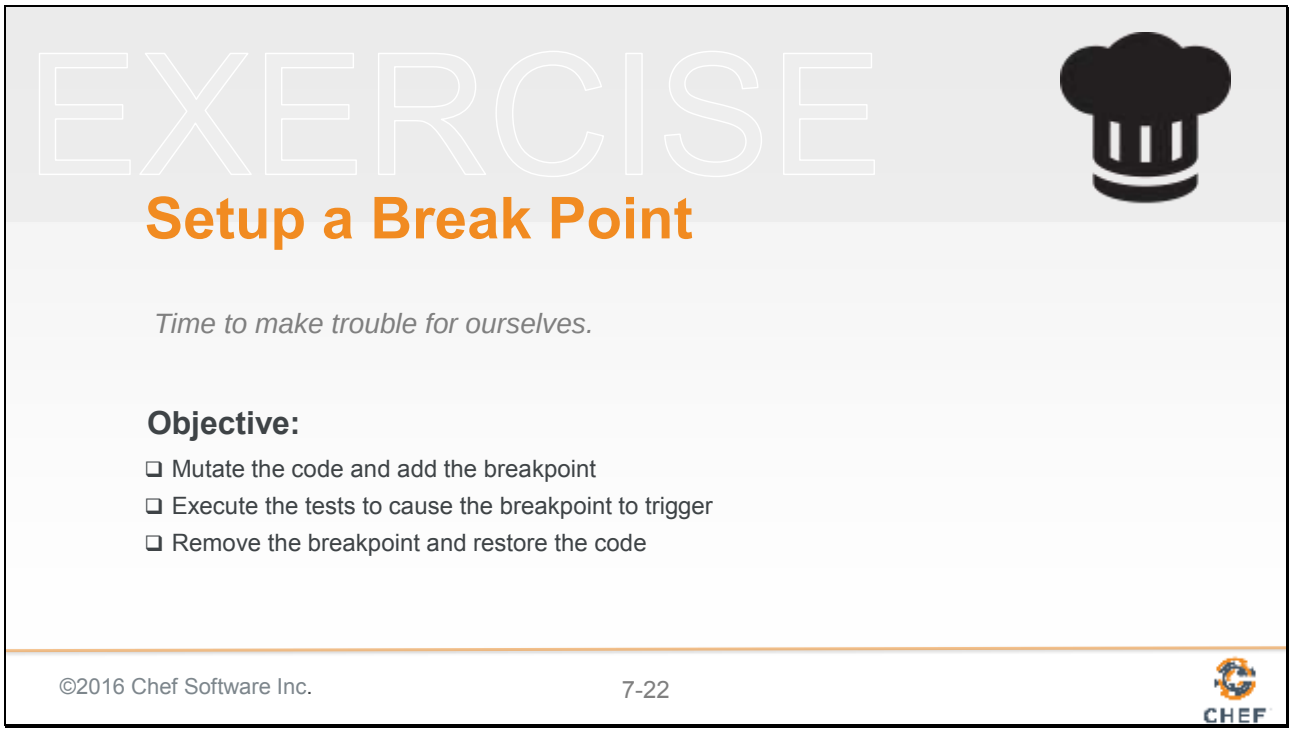
Slide 21



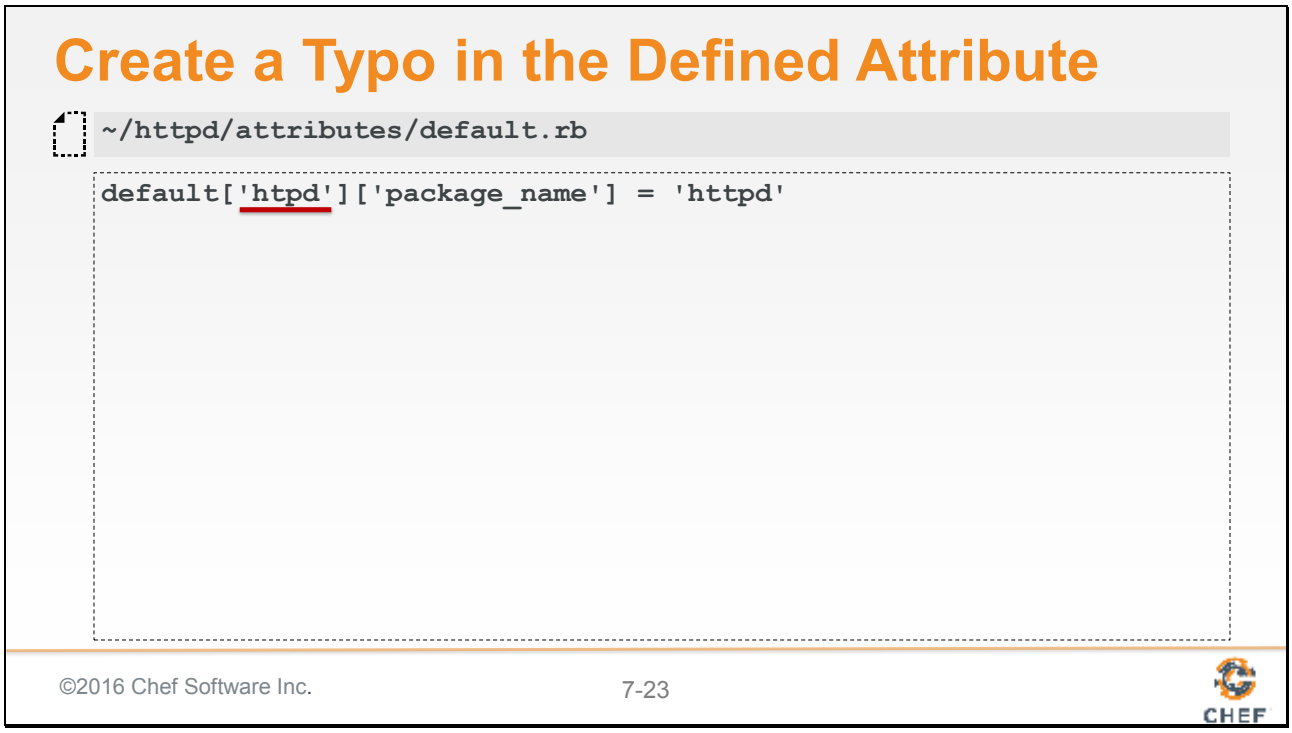
This situation is one in which we want to use a tool called a debugger. Debuggers allow us to set up points where the execution flow will pause and allow us, the user, to interact with the system within the current context of where the execution paused.

Ruby has a well supported debugger project named 'Pry'. 'Pry' is a Ruby gem that is already installed in the Chef Development Kit (Chef DK).

Slide 22

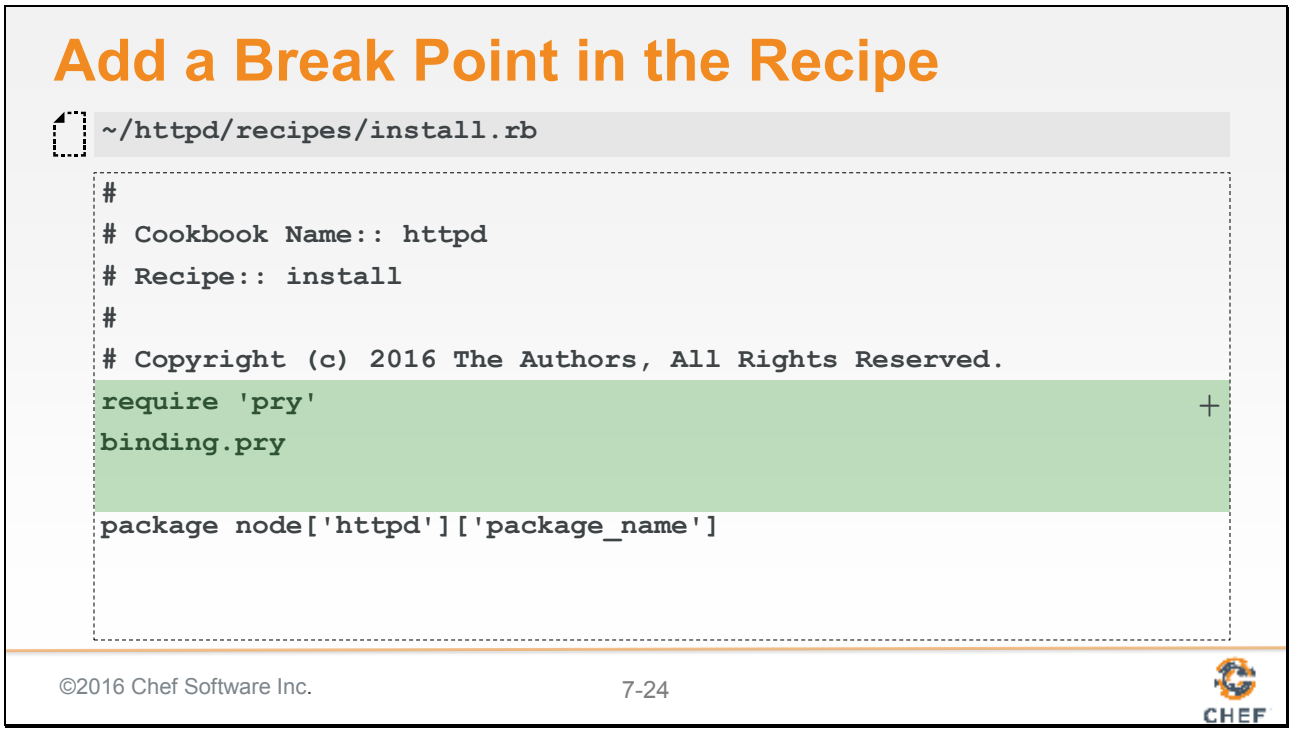


To explore using Pry we need to create an issue for ourselves to troubleshoot. Doing so will allow us to see some of the power of Pry.



This is a simple typo that the examples would catch but when it comes time to find and fix the issue, our eyes may not immediately catch it. We may think the error lies somewhere in the recipe. If we cannot find it we keep running the tests and wondering what is going wrong.

Slide 24

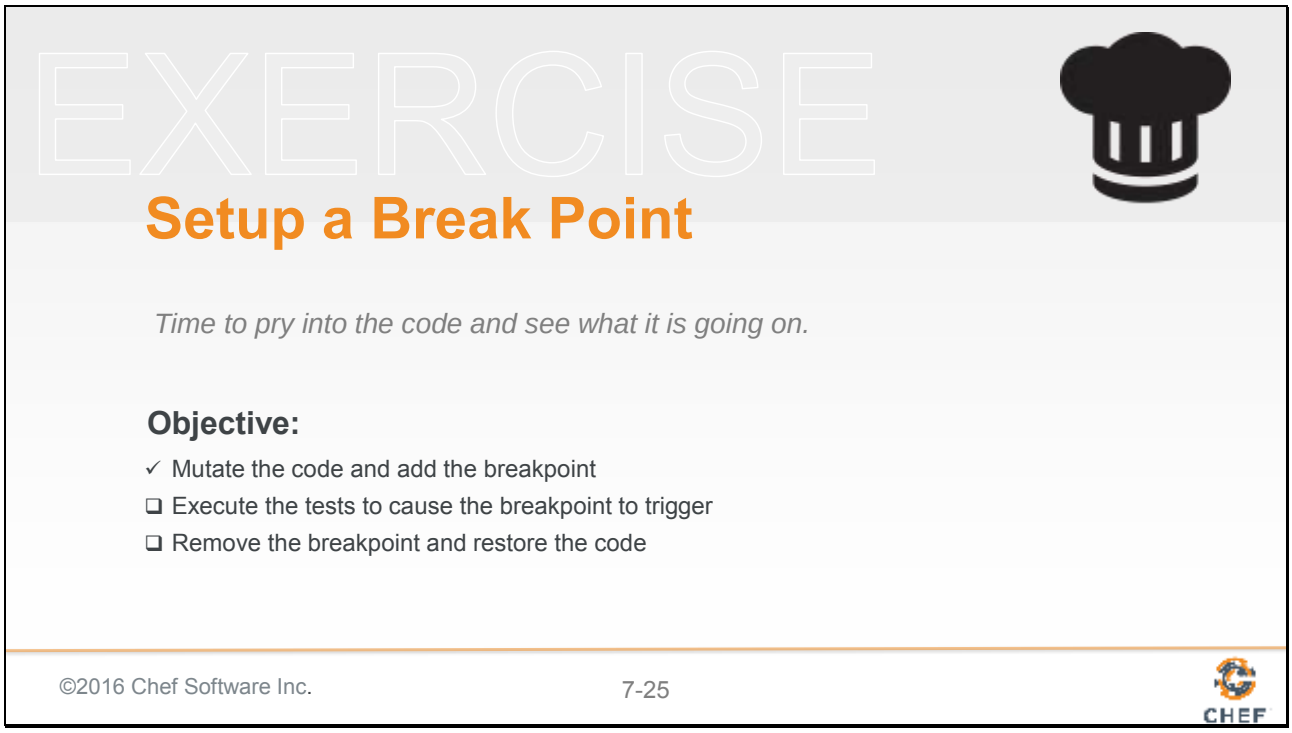


To use Pry you first have to specify a require statement. The require here will look for a file name 'pry' give up on finding it locally and then look for the file inside all of the installed gems.

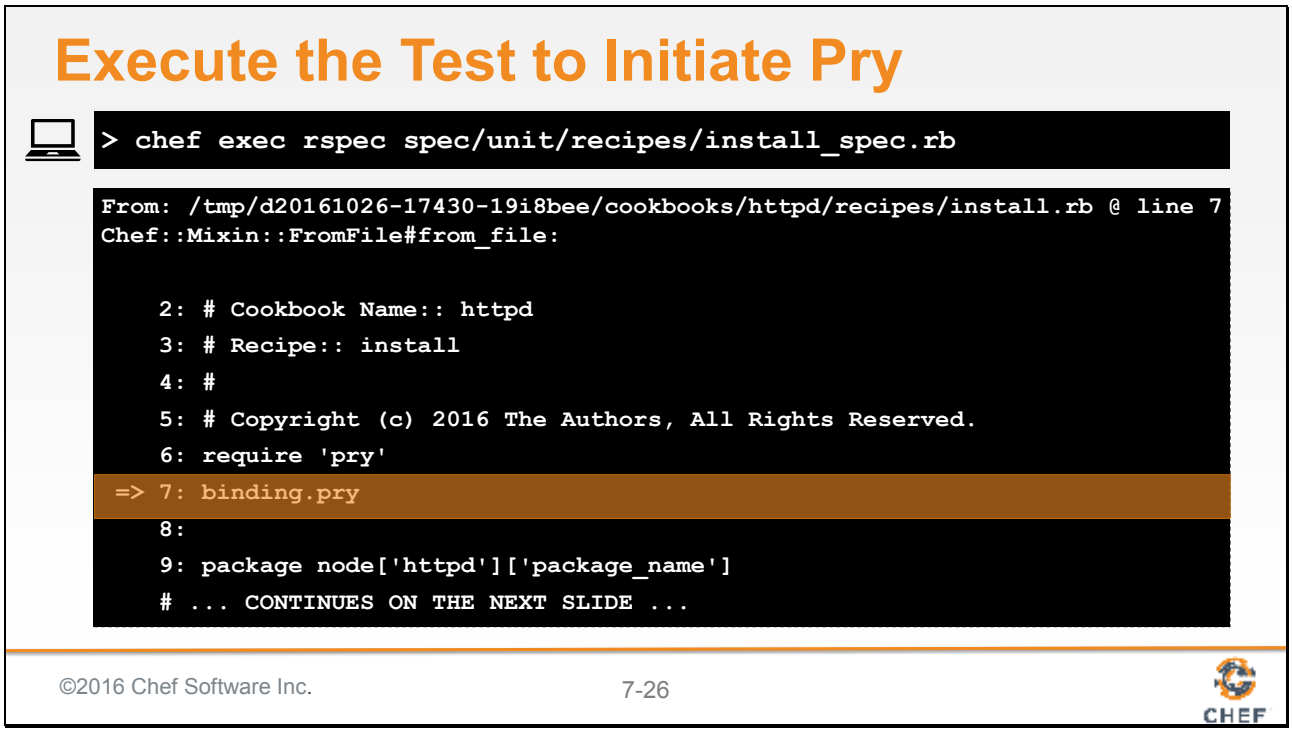
After the Pry code is loaded we access a method named 'binding' and then ask it to run 'pry'. 'binding' is a special method in Ruby that is like gaining access to the DNA of the current context. Pry, after it is loaded, will add the 'pry' method to the binding object to allow us the ability to setup a break point.

Wherever we want to set a breakpoint we can place these two lines.

Slide 25



The breakpoint cannot break itself. We need to execute the code to cause the execution to pause. The best way to do that is execute the tests that we have defined.



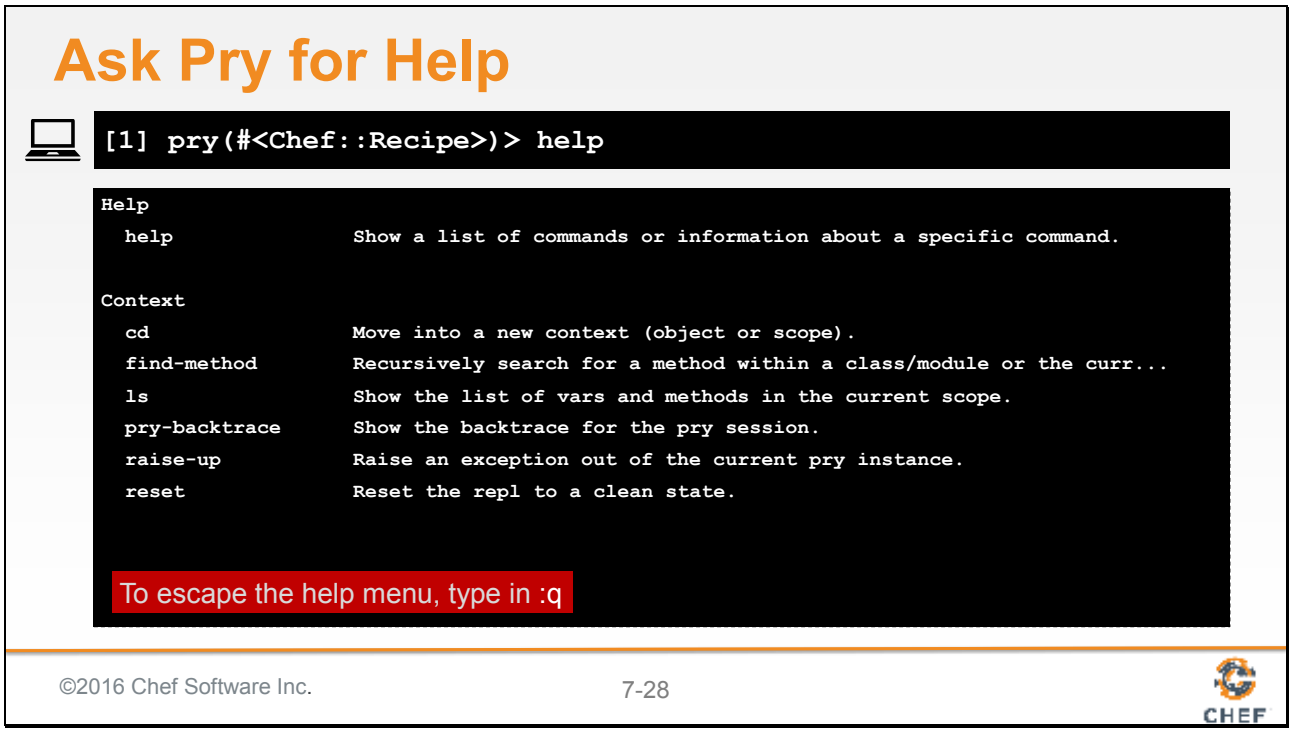
We can execute the tests for the install specification so that it will process that recipe. After a moment of normal execution the flow will pause and you will be shown where in the code the execution has paused. Along the top is the name of the file with the line number where it is paused. Below is a source code listing line-by-line before and after the breakpoint.

Pry Provides an Interactive Prompt

```
> chef exec rspec spec/unit/recipes/install_spec.rb
```

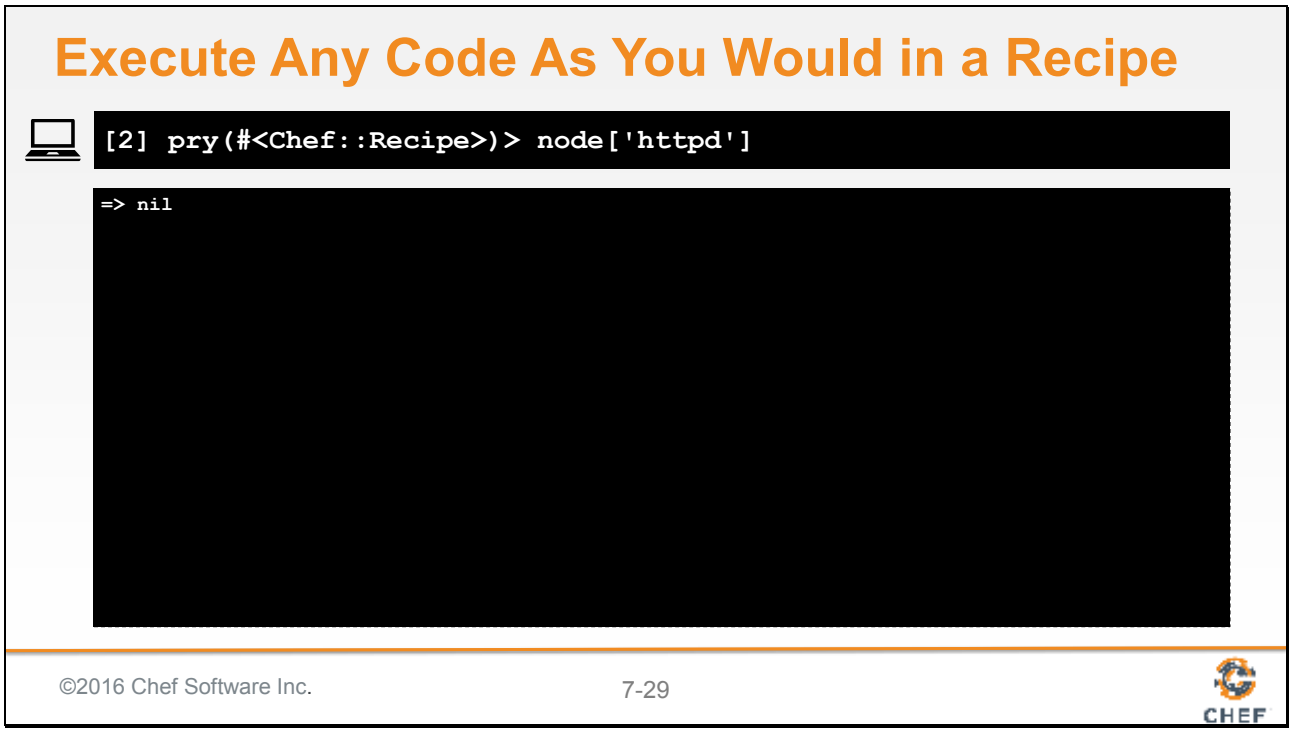
```
    # ... CONTINUED FROM THE PREVIOUS SLIDE ...
    5: # Copyright (c) 2016 The Authors, All Rights Reserved.
    6: require 'pry'
 => 7: binding.pry
    8:
    9: package node['httpd']['package_name']

[1] pry(#<Chef::Recipe>)>
```


Below the summary of the code around the breakpoint is a prompt. Pry launches a Read-Eval-Print-Loop (REPL). At this prompt we can type in a number of commands and any Ruby code.



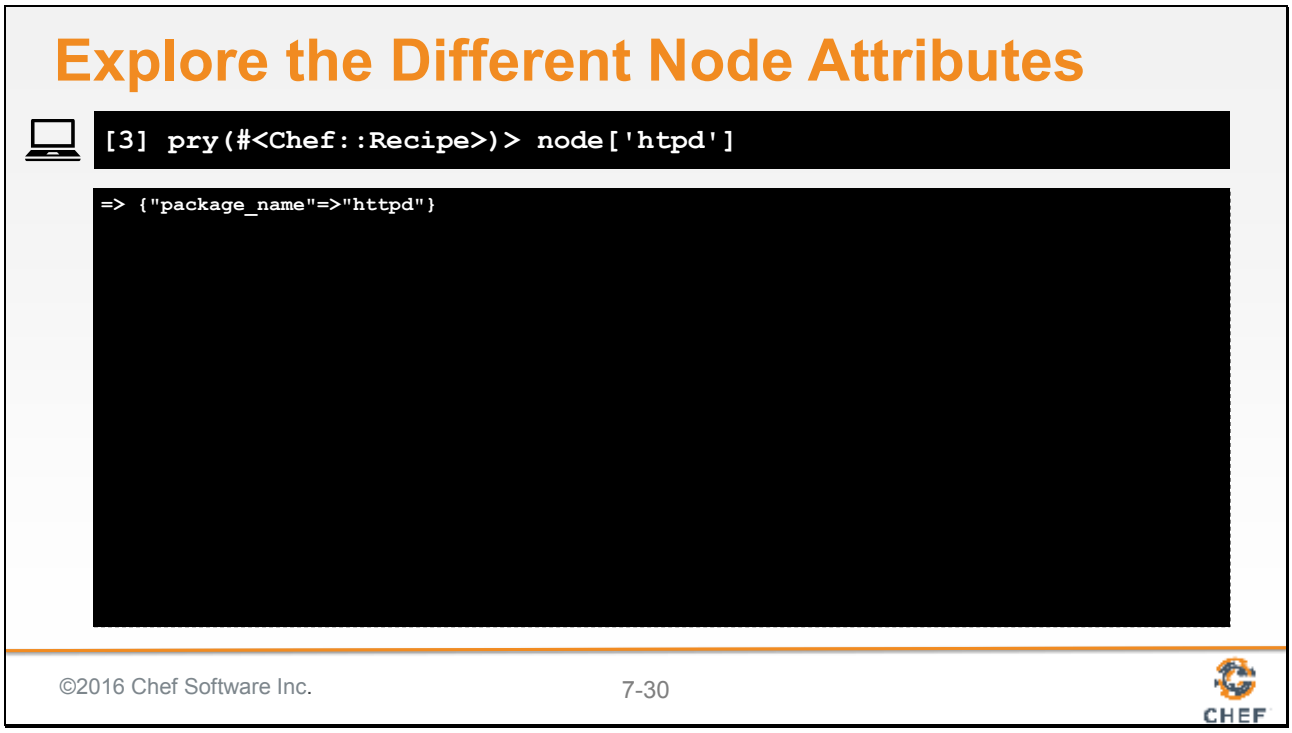
The most important provided by Pry is probably the 'help' command. Within the results of this you will see all the commands available. The help will display in a scrolling page like a Linux man page. To escape out of the help output and return to being able to type in commands you will need to enter the keystrokes ':q'

Instructor Note: This content introduces Pry but will not go into explaining all of the different features.



Back at the prompt you can enter in any code that you would normally write within the recipe. In this case we can start to examine the node object to see that the node object does not have the top-level attribute set as we expected.

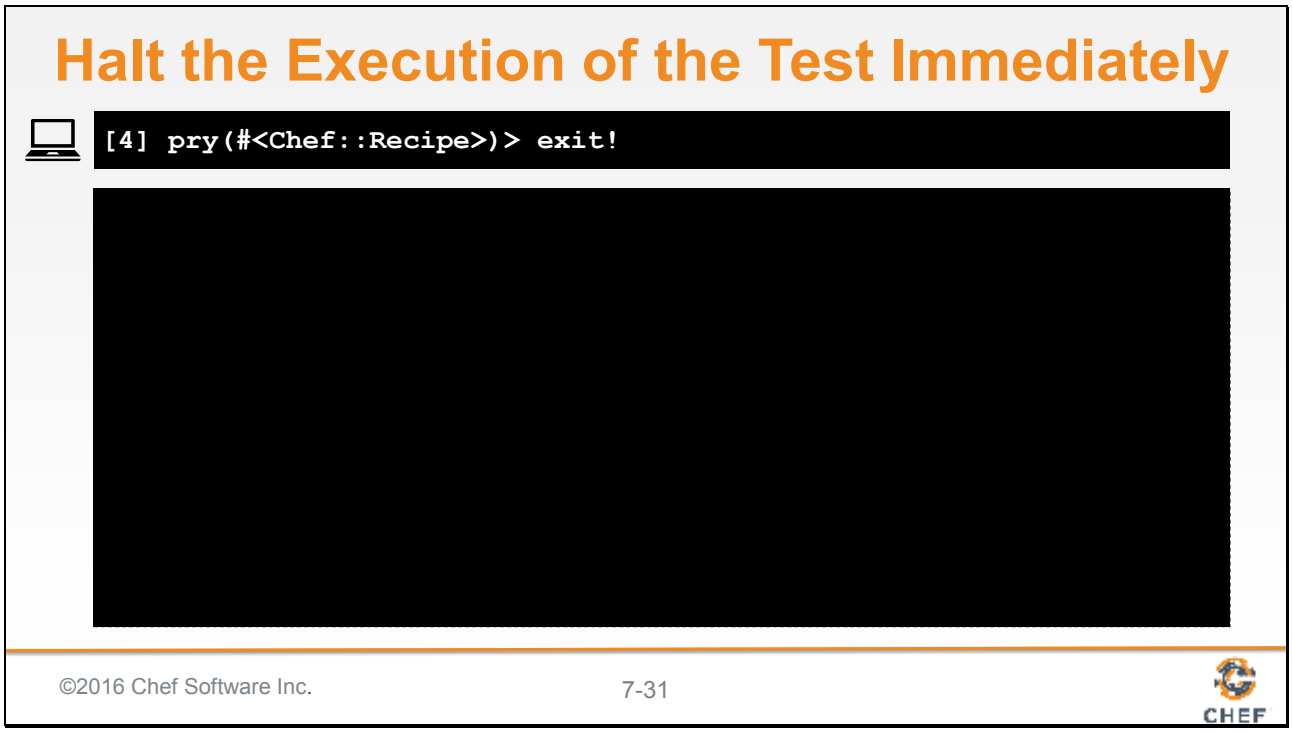
Slide 30



This interactive session allows us to verify the actual state quickly. When it does not match our mental model we can try multiple hypothesizes quickly. Here we may return back to the attribute file and copy the text within the attribute and attempt this again and see what is actually going on.

We see in this example that

Slide 31



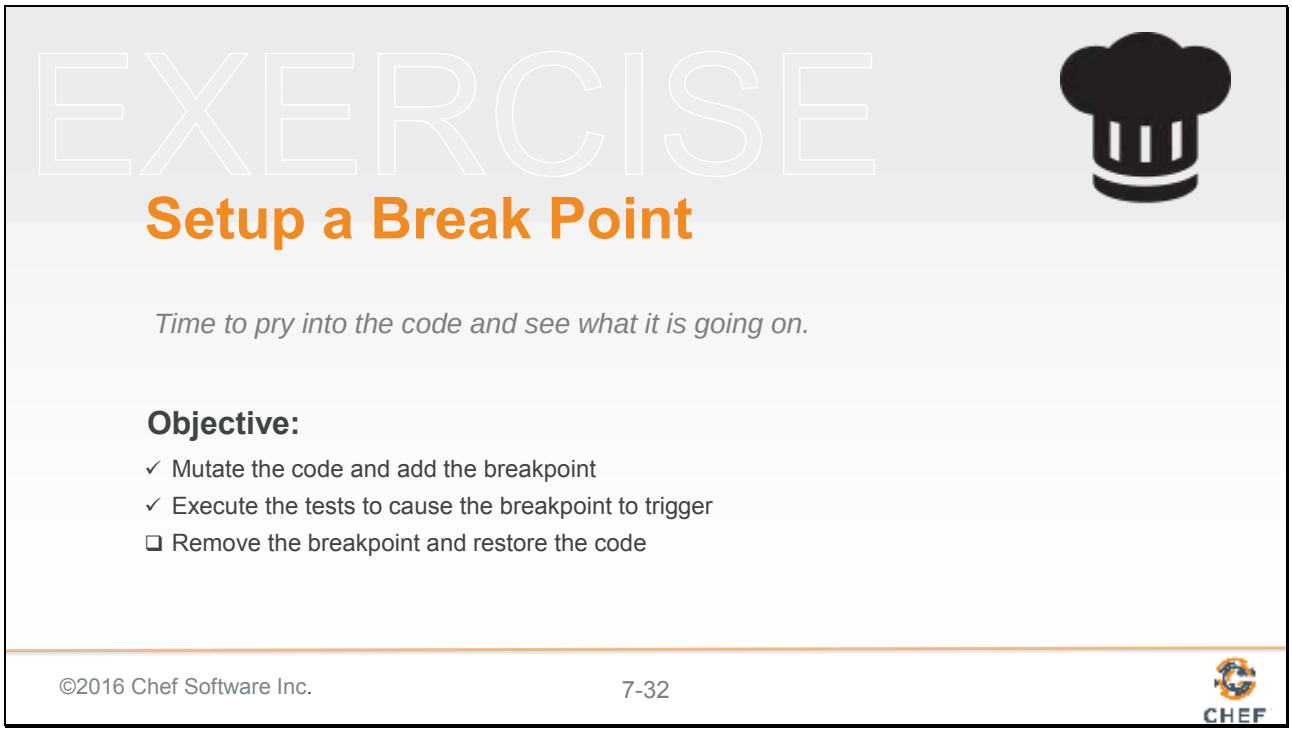
When you are satisfied with what you have discovered it is time to exit. Pry provides two versions of exit:

'exit' which will resume the execution and stop at any other breakpoints along the way.

'exit!' which halts the execution immediately and returns you to your shell.

In this situation we want to halt the execution immediately as we have discovered the issue.

Slide 32



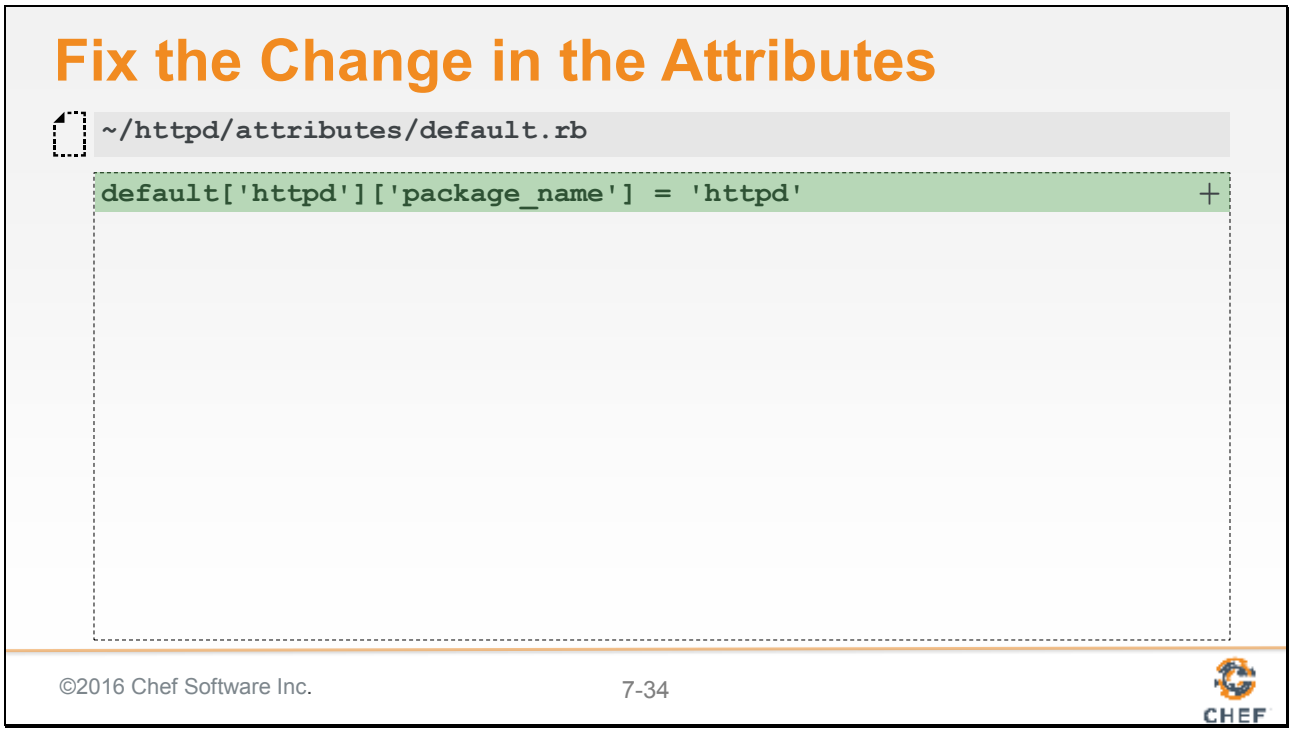
Now that we have discovered the issue in this scenario it is time to remove the breakpoint and restore the attributes code back to its correct state.

Remove the Break Point from the Recipe

~/httpd/recipes/install.rb

```
#
# Cookbook Name:: httpd
# Recipe:: install
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
require 'pry'
binding.pry

package node['httpd']['package_name']
```

Slide 34



This is a simple typo that the examples would catch but when it comes time to find and fix the issue, our eyes may not immediately catch it. We may think the error lies somewhere in the recipe. If we cannot find it we keep running the tests and wondering what is going wrong.

Slide 35

EXERCISE

Setup a Break Point

Time to pry into the code and see what it is going on.

Objective:

- ✓ Mutate the code and add the breakpoint
- ✓ Execute the tests to cause the breakpoint to trigger
- ✓ Remove the breakpoint and restore the code

CHEF

This small exercise focused on a small subset of what is possible with Pry. It is a powerful tool that will aid you in understand the execution of the system much faster than tests alone.

Slide 36

LAB

Refactor Remaining Resources

- ❑ Refactor the resource to use a Node attribute
- ❑ Execute the tests and verify the tests fail
- ❑ Add the new Node attribute
- ❑ Execute the tests and verify the tests pass

BONUS: Use pry to verify that the attribute has been set.

❖ **Repeat this series of steps for the configuration recipe and service recipe**

CHEF

Now it is your turn. Two recipes remain that I want you to refactor to use attributes. Follow the same workflow you used here. As a bonus try using Pry again to reinforce setting it up and navigating through the execution flow with it.

Instructor Note: Allow 10 minutes to complete this exercise

Slide 37

Update the Recipe to use the Node Attribute

~/httpd/recipes/service.rb

```
#
# Cookbook Name:: httpd
# Recipe:: service
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
service node['httpd']['service_name'] do
  action [:enable, :start]
end
```

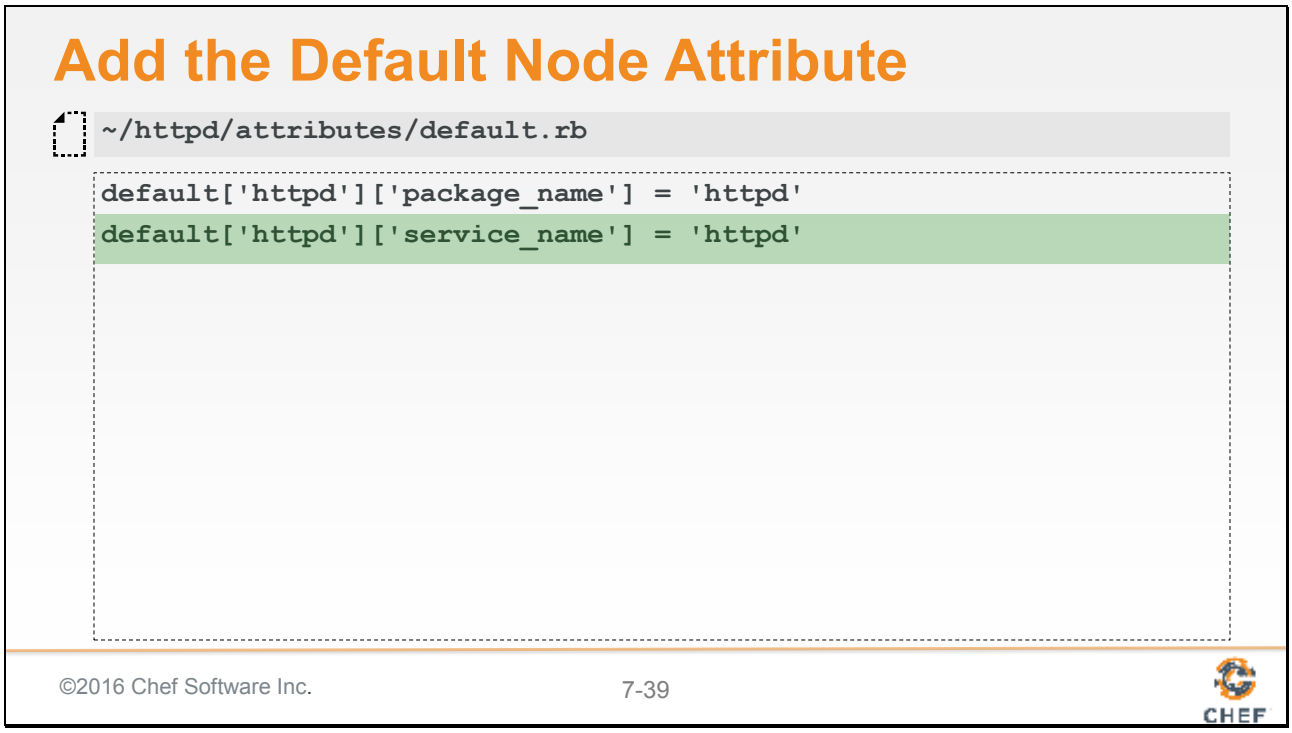

Let's review the refactoring of the service resource. You returned first to the service resource in the service recipe and specify a node attribute that will give you the service name.

Execute the Tests to See it Fail

```
> chef exec rspec spec/unit/recipes/service_spec.rb

FFF

Failures:

  1) httpd::service When all attributes are default, on an unspecified
platform converges successfully
     Failure/Error: expect { chef_run }.to_not raise_error
       expected no Exception, got #<NoMethodError: undefined method `[]' for
nil:NilClass> with backtrace:
         # /tmp/d20161027-27872-9rctn8/cookbooks/httpd/recipes/service.rb:6:in
`from_file'
```

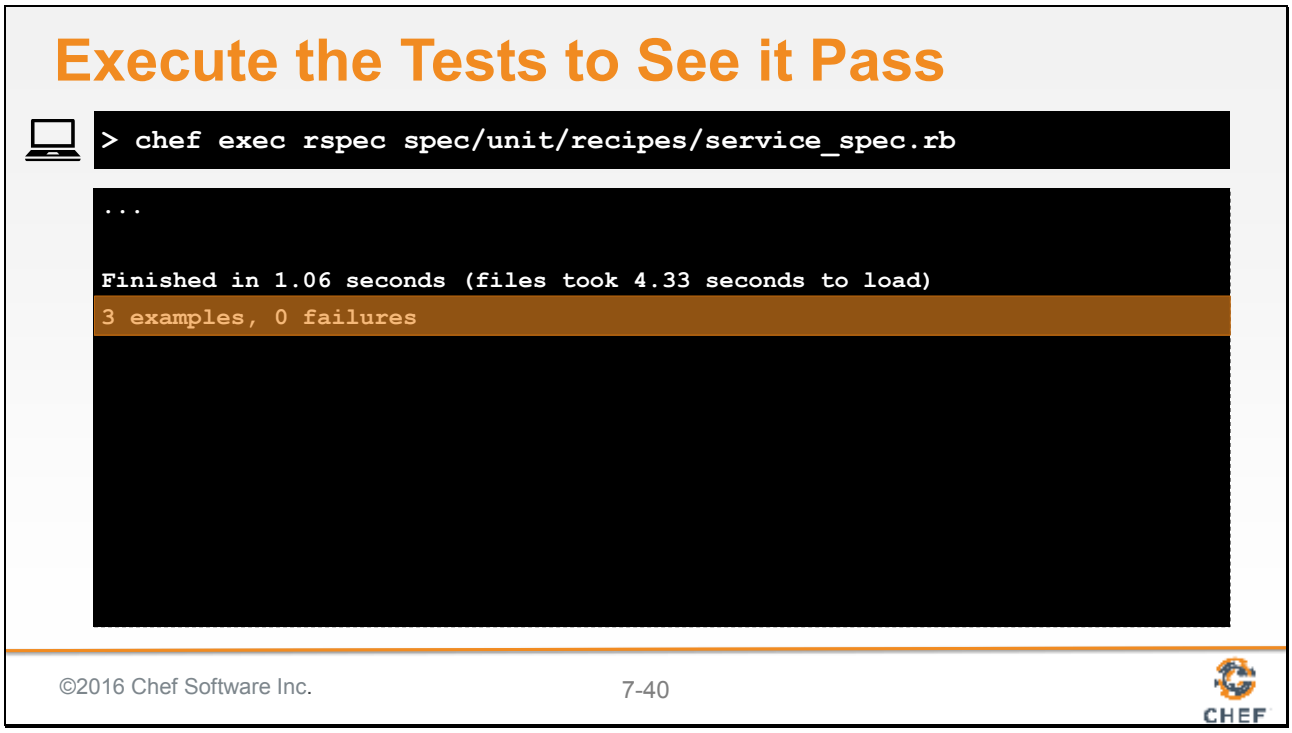

You executed the tests against all the recipes or the specific service recipe. A large set of errors appear as we saw last time. The error is telling us to define the node attribute.

Slide 39



You opened the default attributes file up and defined the new node attribute at the default level.

Slide 40



You executed the tests again and saw all the expectation have been met successfully.

Slide 41

LAB

Refactor Remaining Resources

- ✓ Refactor the resource to use a Node attribute
- ✓ Execute the tests and verify the tests fail
- ✓ Add the new Node attribute
- ✓ Execute the tests and verify the tests pass

BONUS: Use pry to verify that the attribute has been set.

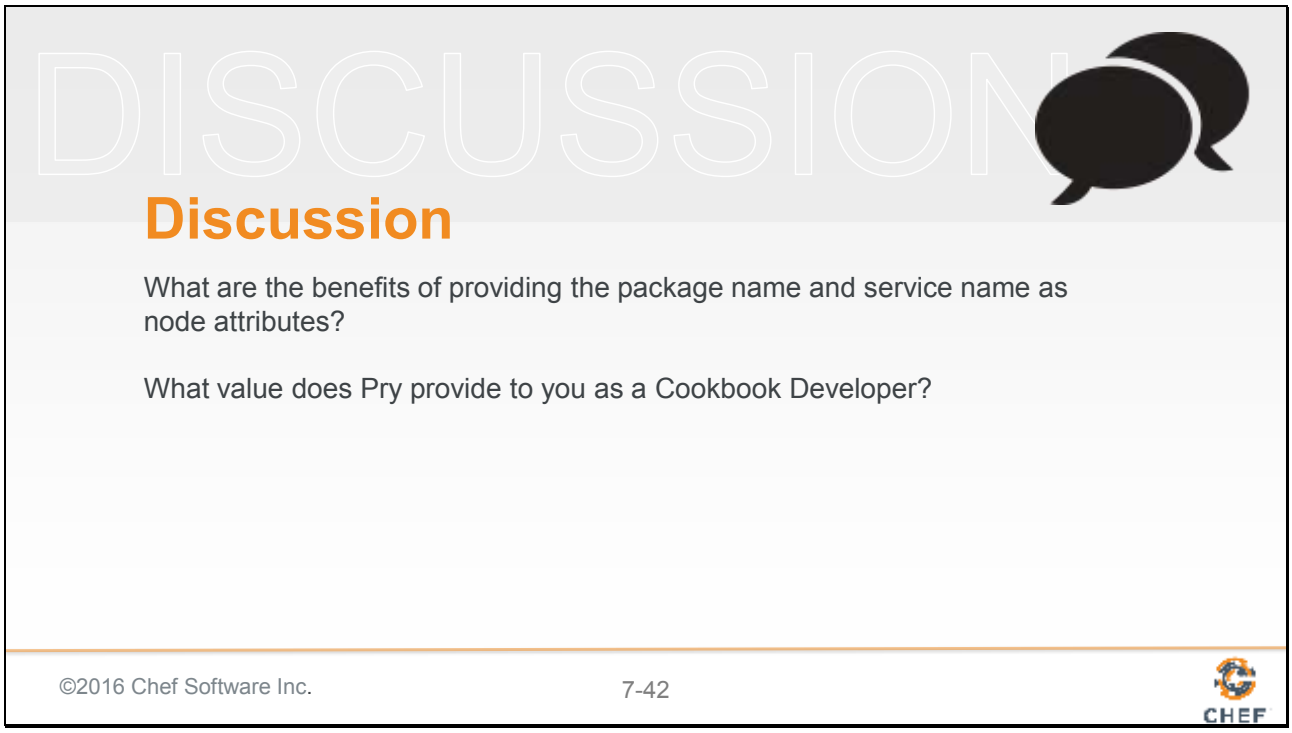
❖ **Repeat this series of steps for the configuration recipe and service recipe**

Congratulations. Now you have completely refactored the resources in the cookbook to use node attributes.

Let's have a discussion.

Instructor Note: We did not review the configuration recipe

Slide 42

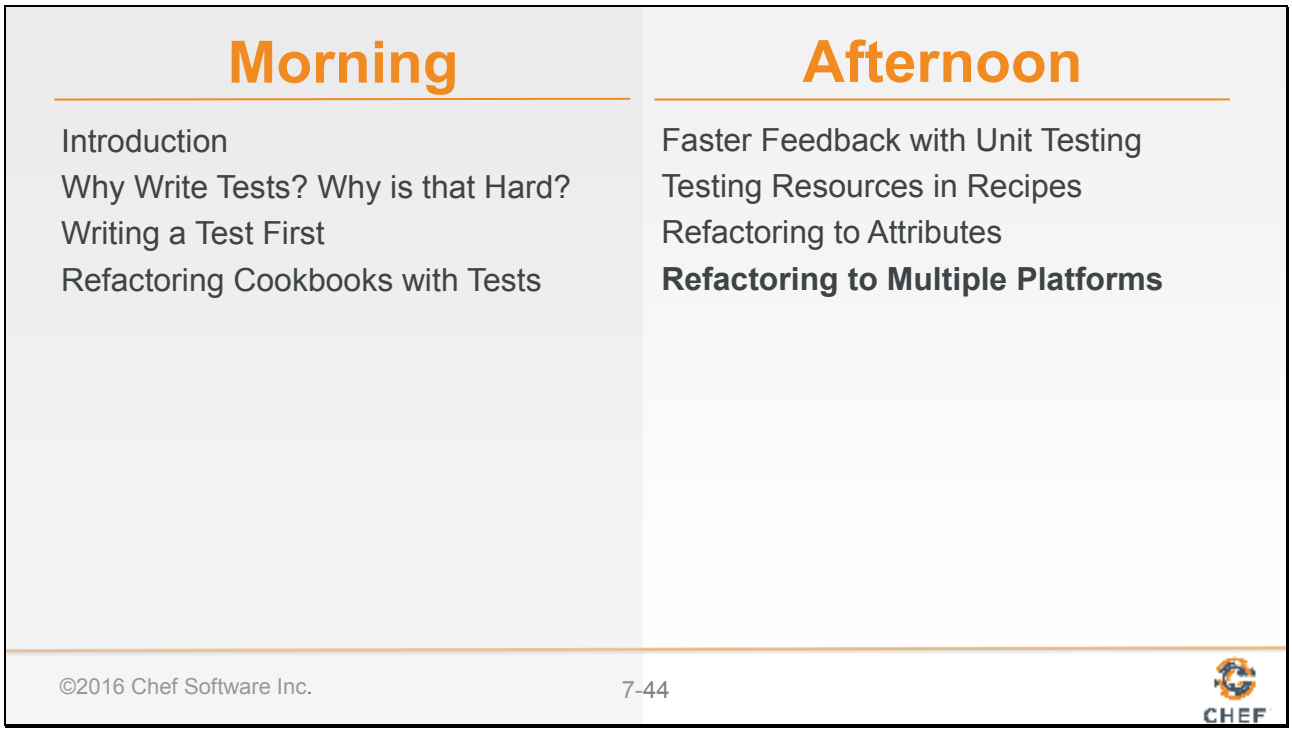


Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 43

Before we complete this section, let us pause for questions.

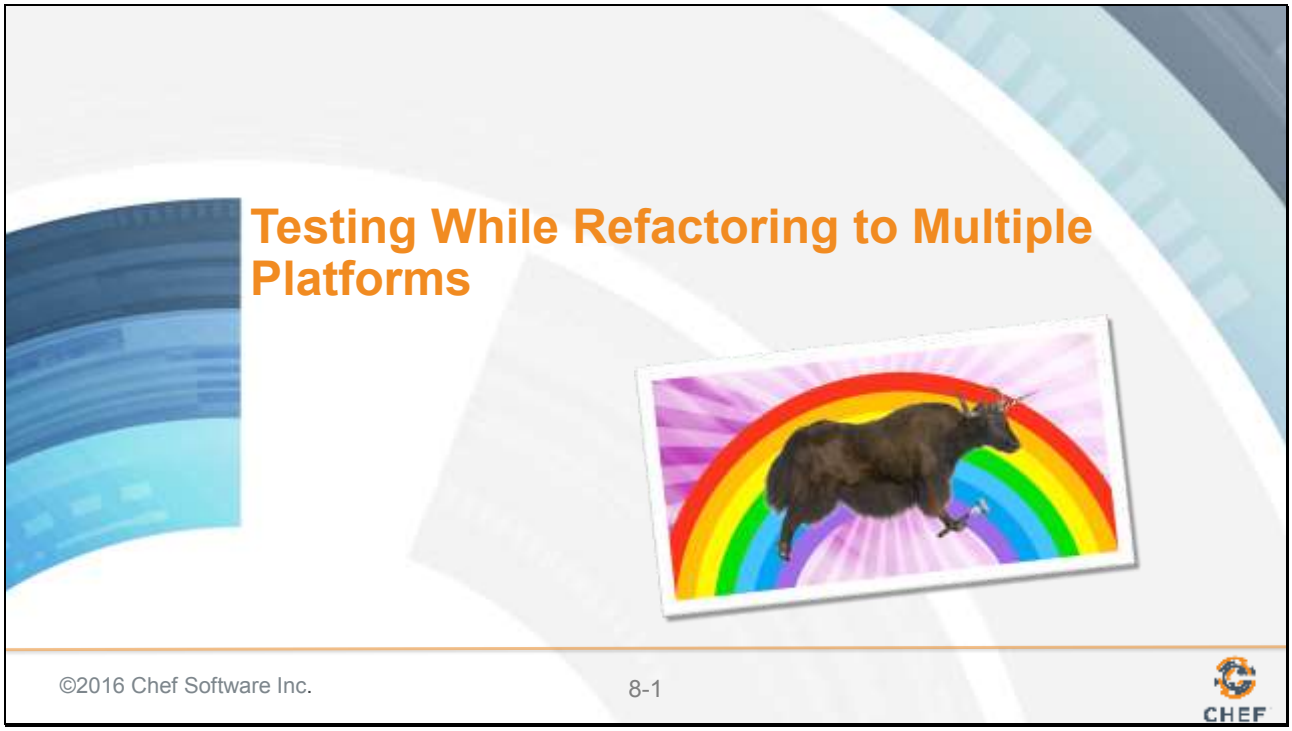
Slide 44



With the resources now using node attributes we are ready to explore the last section which will challenge us to expand the scope of this cookbook to support multiple platforms.

Slide 45

8: Testing While Refactoring for Multiple Platforms



When we started developing this cookbook I told you that we were going continue to refactor this cookbook until it supported multiple platforms. We could have started with that goal. Instead we started small. One test. One recipe. Refactor. Add more tests. Refactor. This process allowed us to deliver a reliable cookbook in confident way. But testing was not the only thing that aided us in building this cookbook.

Instrumental to software development and test-driven development is learning how to divide the work into these small increments. Small, deliverable, verifiable steps are essential to developing code with confidence. Now that you have seen and experienced the Test Driven Development (TDD) workflow and understand the basics, the real work that lay before you is to understand how to find these divisions in the requirements you are given.

This was a hand-picked experience. That moves we made may have seemed contrived. As with any knowledge transfer the best we can do is give you a model to play with and hope the forms hold true when it comes time for you to solve a problem with real requirements.

Slide 2

Objectives

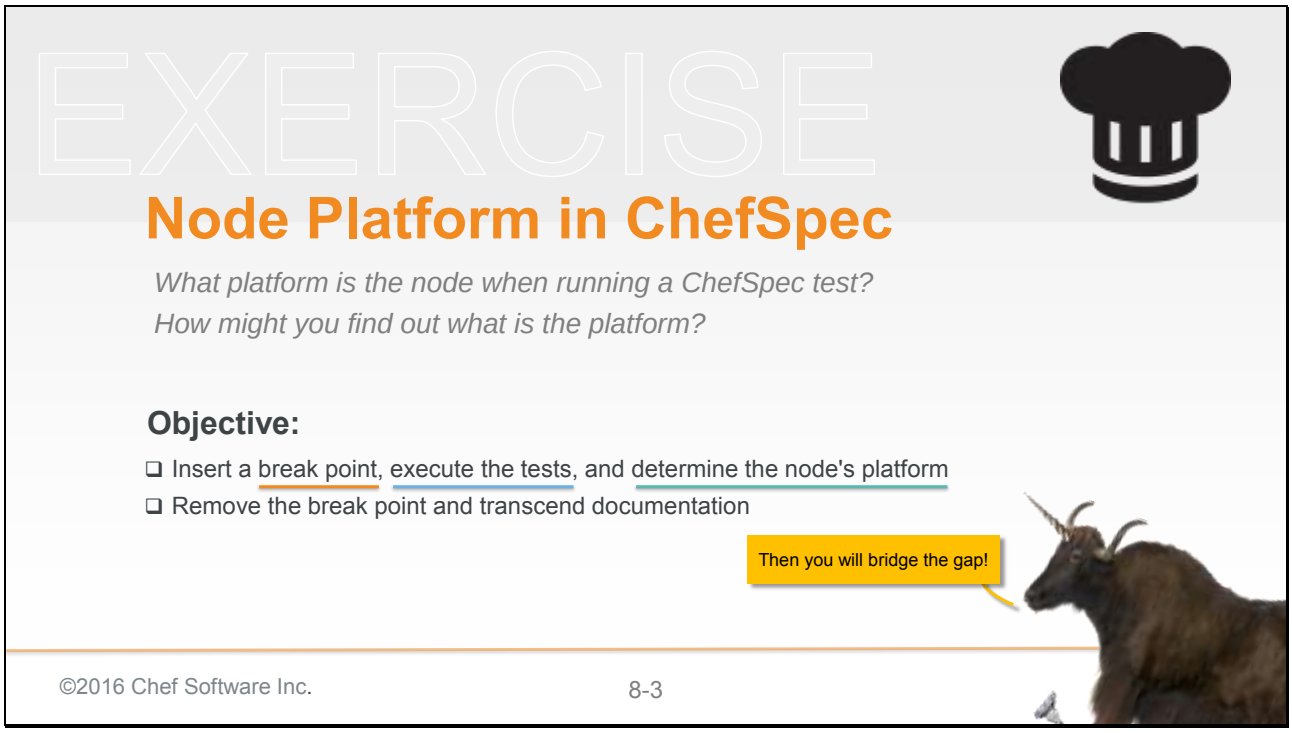
After completing this module, you should be able to:

- Define expectations for multiple platforms
- Implement a cookbook that supports multiple platforms

CHEF

In this module you will learn how to define expectations for multiple platforms and implement a cookbook that supports multiple platforms.

Slide 3



In this module we are going to develop solution in the opposite of the way we started. Instead of approaching this problem from the outside-in we are going to build it inside-out.

To do that means we are going to leverage the specifications we have written that validate the resources within our recipe. But before we do we need to gather some information that is important. Like the name of the platform we are using?

We could attempt to solve this problem by looking for documentation or a general search on the Internet. Instead we will ask the one source that knows the best: the executing code itself.

Slide 4

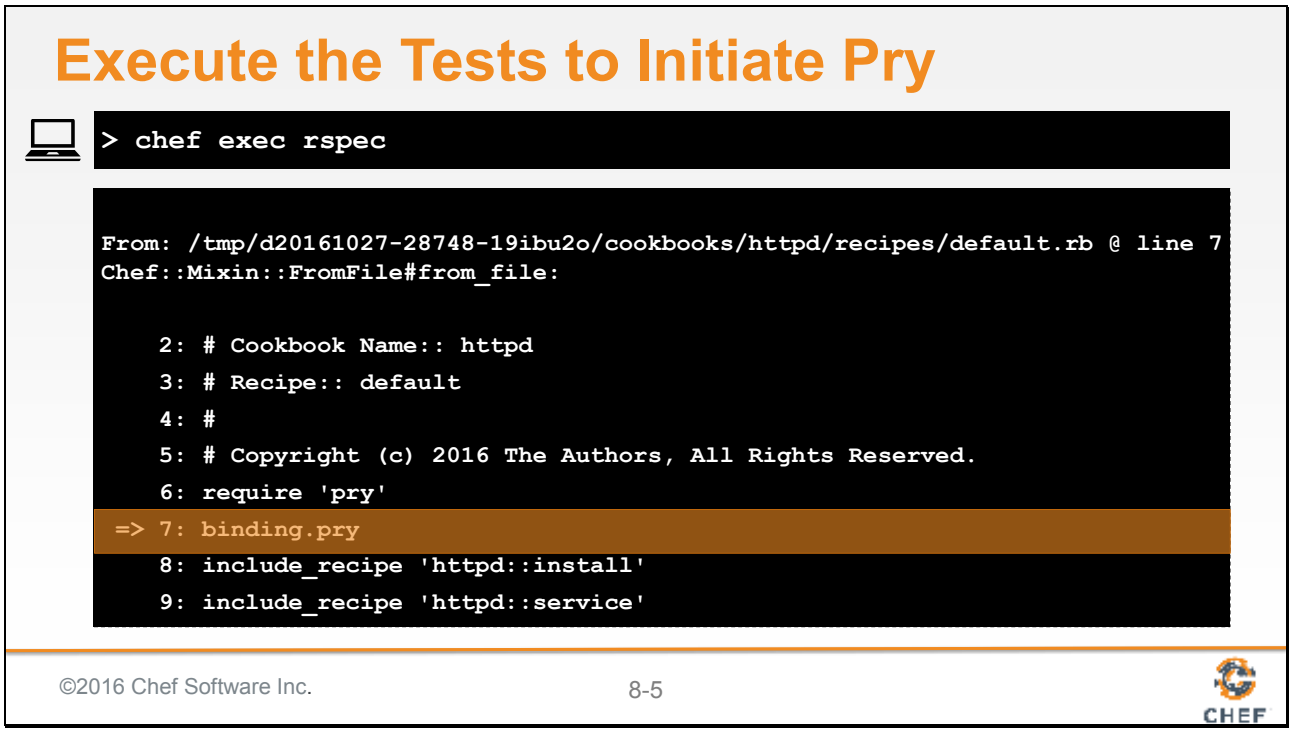
Add a Break Point to the Default Recipe

~/httpd/recipes/default.rb

```
#
# Cookbook Name:: httpd
# Recipe:: default
#
# Copyright (c) 2016 The Authors, All Rights Reserved.
require 'pry'
binding.pry
include_recipe 'httpd::install'
include_recipe 'httpd::configuration'
include_recipe 'httpd::service'
```

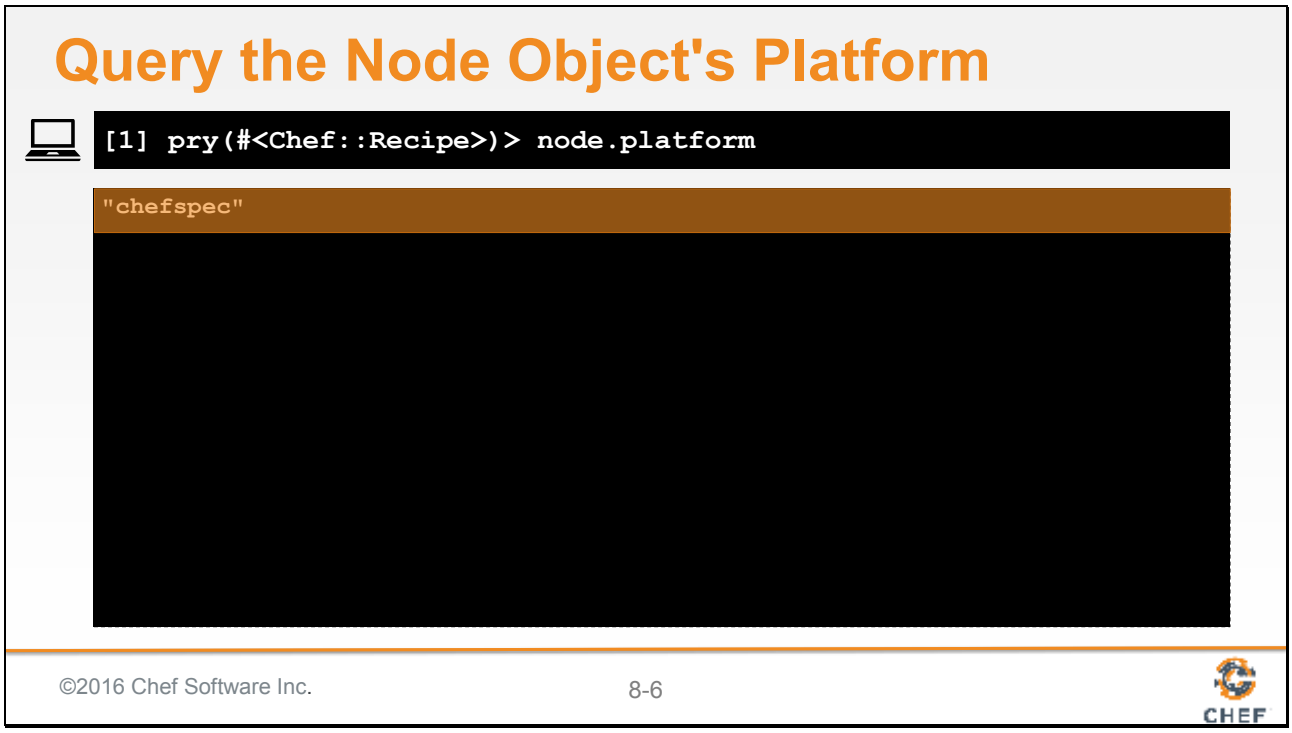

To understand the platform of the node we simply need to set a break point in one of the recipes or the attributes file.

Slide 5



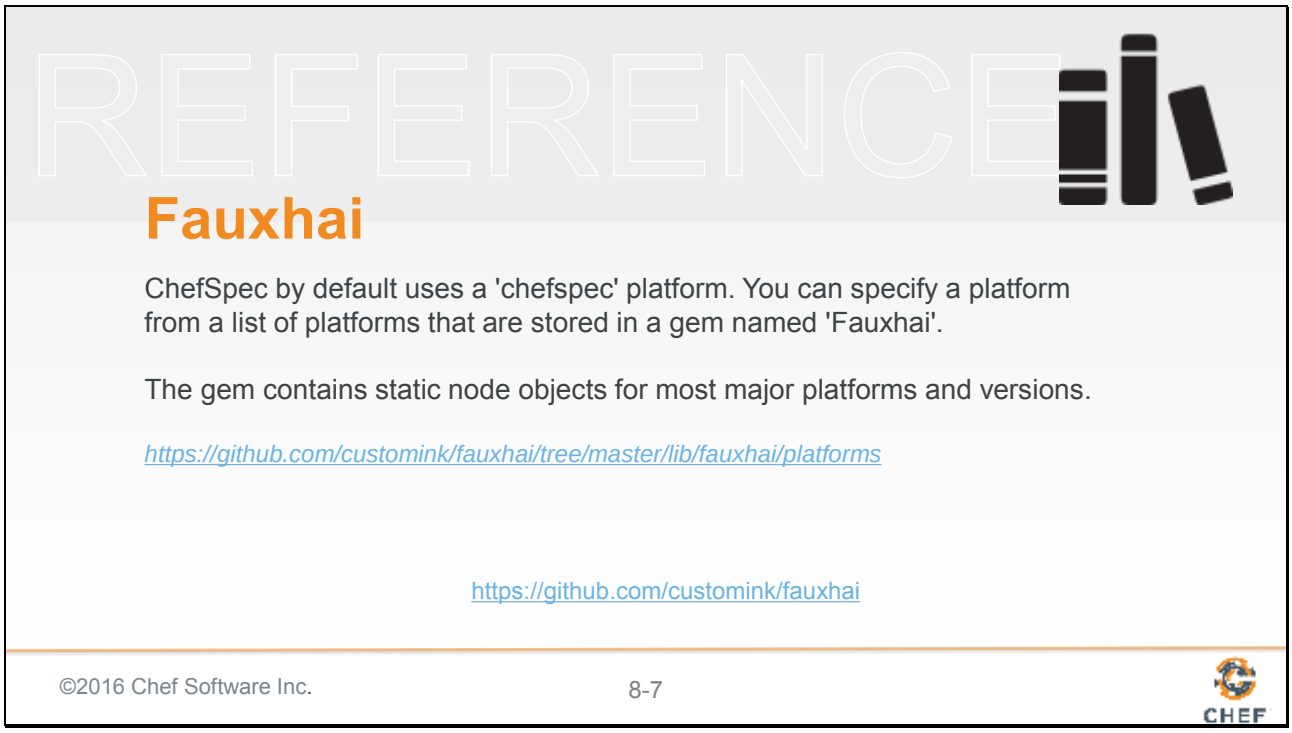
Execute the tests.

Slide 6



And then query the platform of the node object. The results should tell you that the platform for the node object in the ChefSpec environment is 'chefspec'.

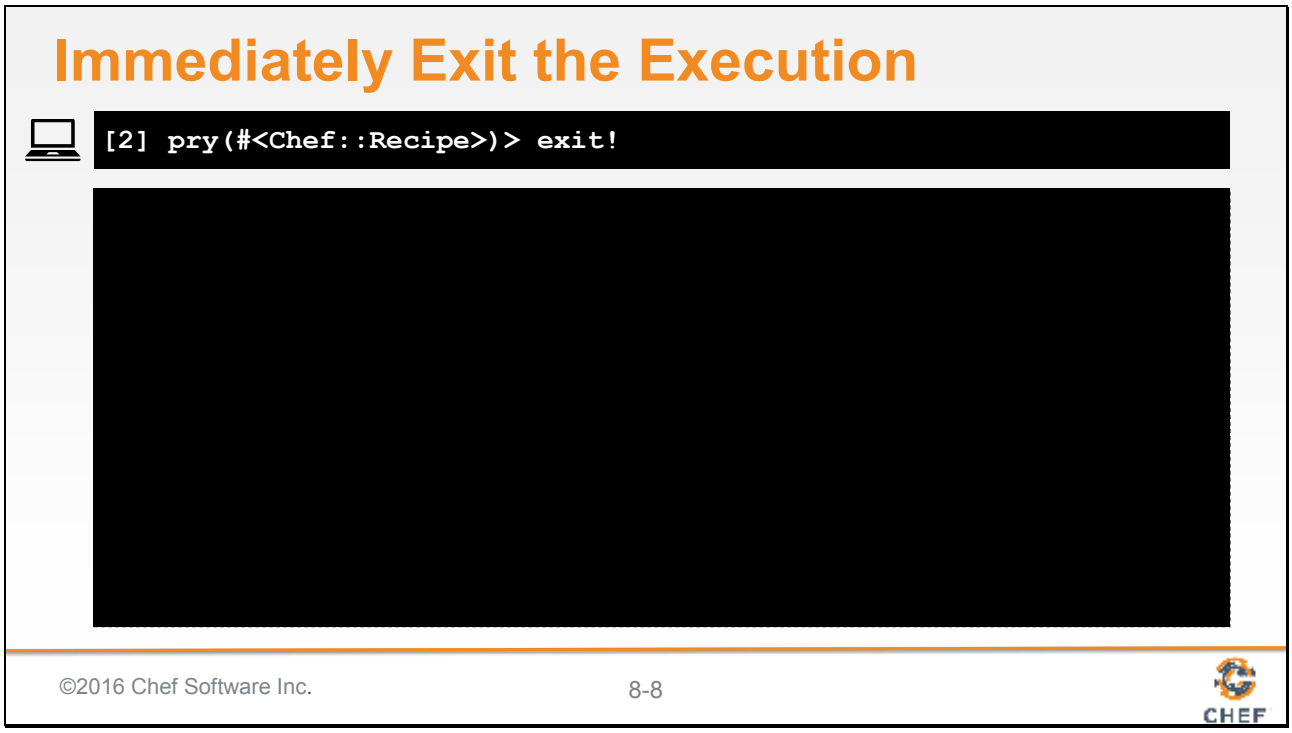
Slide 7



The 'chefspec' platform is set by the ChefSpec gem. The platform has gone unspecified and this is what ChefSpec defaults to use. Now that we care about the platform we need to learn about another gem named Fauxhai. ChefSpec employs Fauxhai to provide fake node object data for various platforms.

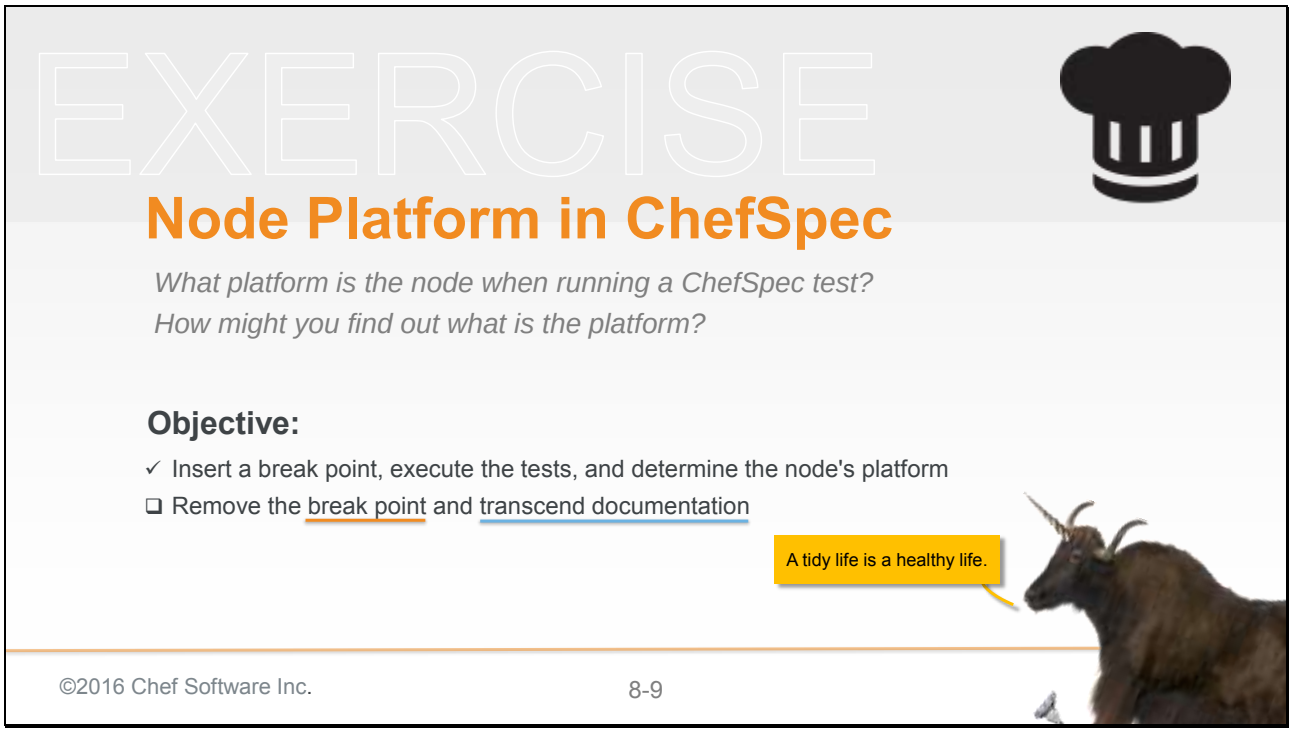
These platforms and their various versions are defined in the gem itself. Essentially the gem, at the time of writing this, contains a large number of JSON files which hold the node object results on each specific platform and version it supports. The best way to learn what platforms are provided is to read the source code in the Fauxhai repository.

Slide 8



Now that we know the platform it is time to exit the execution.

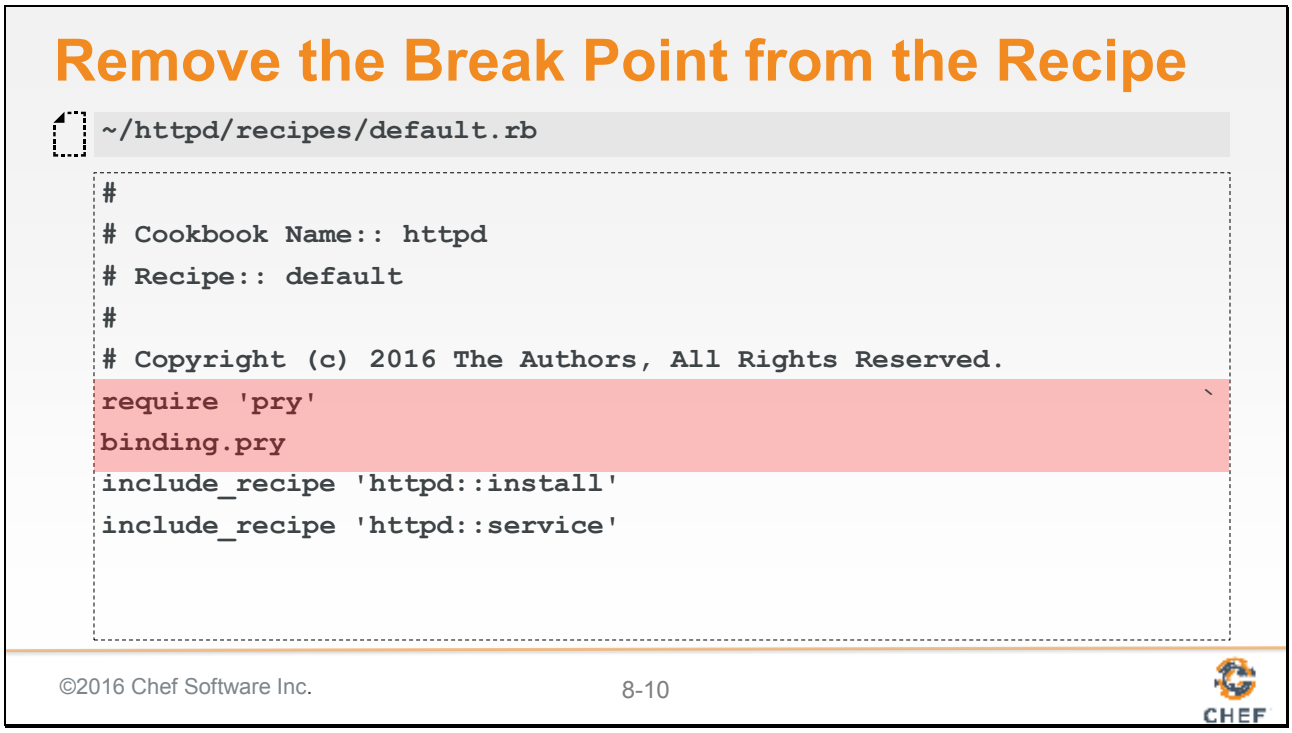
Slide 9



Using Pry we were able to learn something about the system without having to rely on documentation. To understand the available platforms you have to rely on reading the source code.

Learning this powerful skill of gathering details will help you solve mysteries and provide more details and queries when searching for help on the Internet. The better you can get at understanding when to employ Pry and how to use it will eventually have you using documentation less and using executing code and source code more.

Instructor Note: Finding out which platforms and versions ChefSpec supported alluded me when first working with the project. There is some mention in the ChefSpec README but I believe I found myself diving into source code and stumbling upon the Fauxhai code. This is something that would be great to show to show learners if you are capable of figuring that out.



It is a good habit to clean up this break points. Leaving them around has a nasty habit of pausing the execution of a run you want to see complete uninterrupted.

Slide 11

EXERCISE

Node Platform in ChefSpec

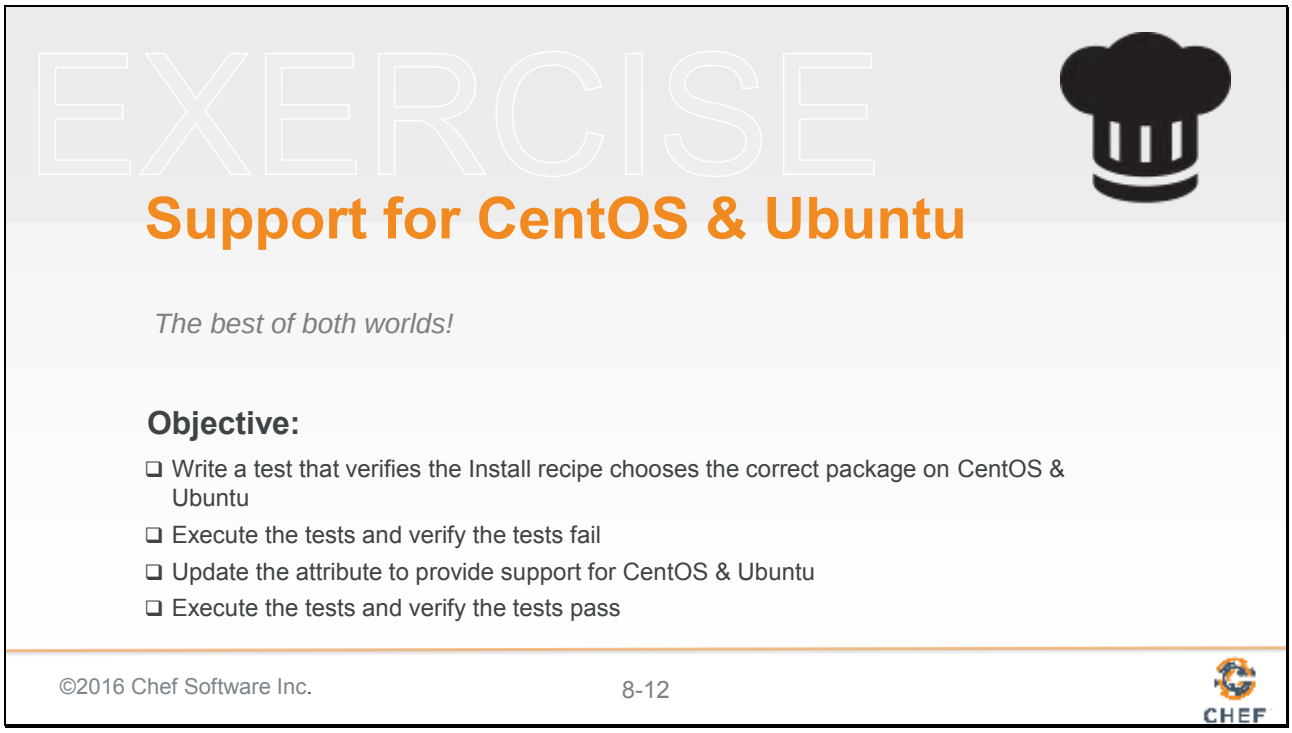
What platform is the node when running a ChefSpec test?
How might you find out what is the platform?

Objective:

- ✓ Insert a break point, execute the tests, and determine the node's platform
- ✓ Remove the break point and transcend documentation

Now that we know the environment it is time to get to work on defining those new examples for the new platform that we want to support.

Slide 12



Together let's walk through refactoring the cookbook's install recipe. Like we have done before. When we are done it will be your turn to implement the solution for the remaining recipes.

Slide 13

Update the Context to be Platform Specific

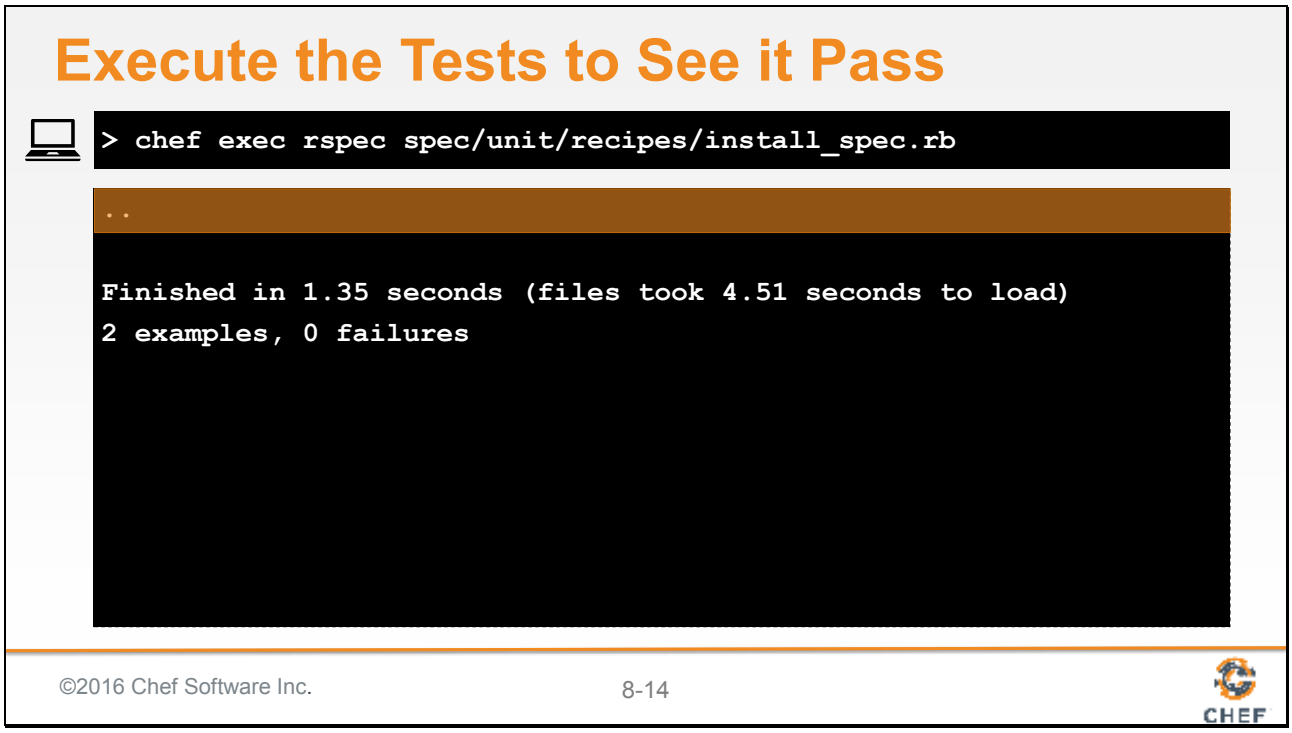
~/spec/unit/recipes/install_spec.rb

```
describe 'httpd::install' do
  context 'When all attributes are default, on CentOS' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.7')
      runner.converge(described_recipe)
    end

    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end

    it 'installs the appropriate package' do
      expect(chef_run).to install_package('httpd')
    end
  end
  # ... SPECIFICATION CONTINUES ON THE NEXT SLIDE ...
```


First we will start by updating our current specification. The context up to this point has been 'on an unspecified platform'. We want to instead state that these first two examples are for the CentOS platform. That change is purely cosmetic.

The change that matters is the one in which we provide new parameters to the ServerRunner class initializer that state the specific platform and version we are interested in verifying against. If we specify an unsupported platform or platform version we will see an error when the tests execute. This is again why it is important to review the Fauxhai project.

Slide 14



Because we made changes the original expectations it might be a good moment to execute the tests and ensure that everything is still working for the CentOS platform.

Slide 15

Add a Second Context for Another Platform

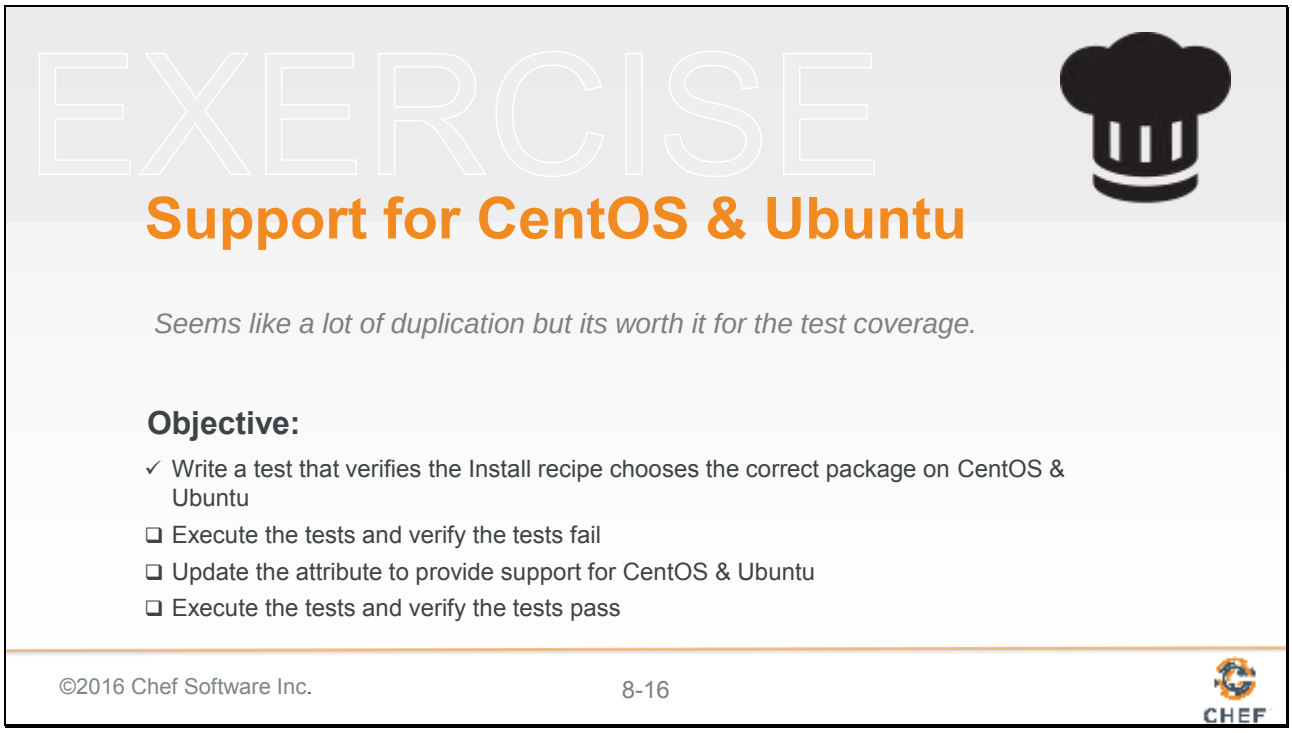
~/spec/unit/recipes/install_spec.rb

```
  # ... CONTINUED FROM THE PREVIOUS SLIDE ...
  context 'When all attributes are default, on Ubuntu' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '14.04')
      runner.converge(described_recipe)
    end

    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end

    it 'installs the necessary package' do
      expect(chef_run).to install_package('apache2')
    end
  end
end
```

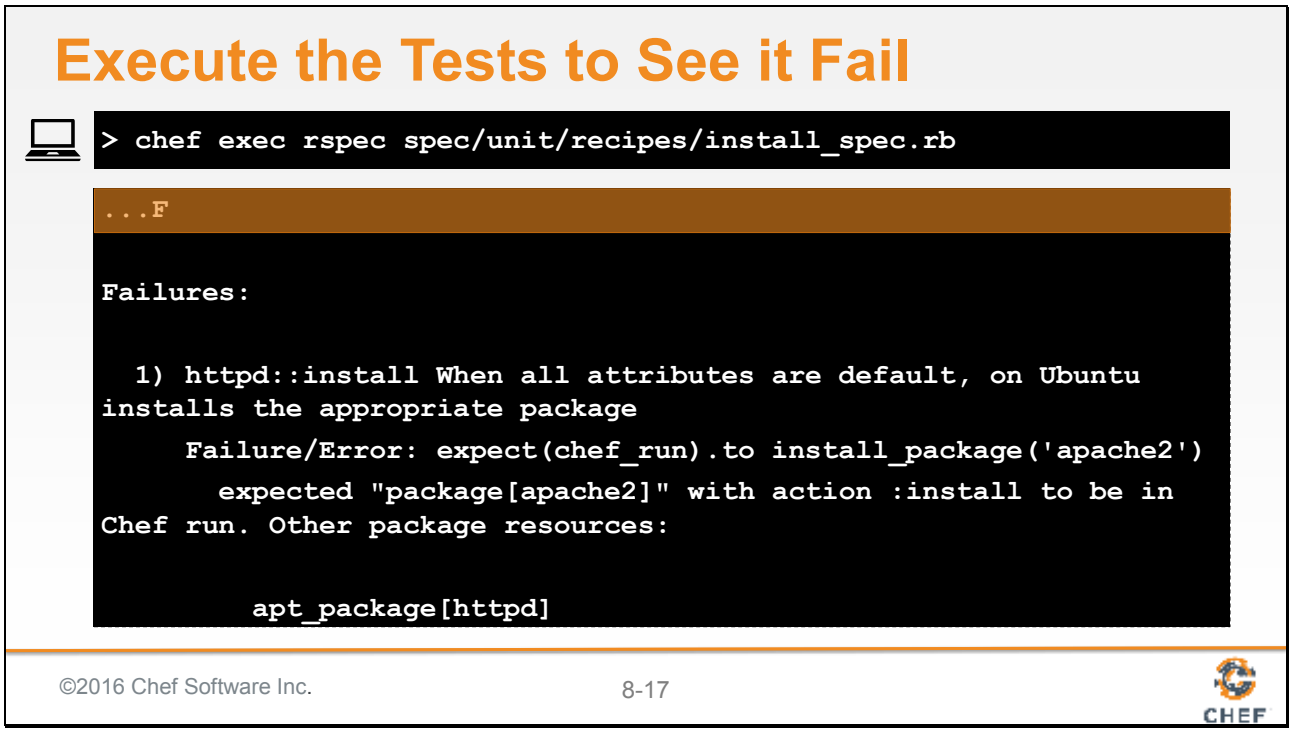

Now return to the specification file and alongside CentOS example group it is time to define the example group that will contain the examples for the Ubuntu 14.04 platform.

The format is nearly identical between these two example groups save for the context, the parameters specified to the ServerRunner initialization, and the name of the necessary package to install.

Slide 16

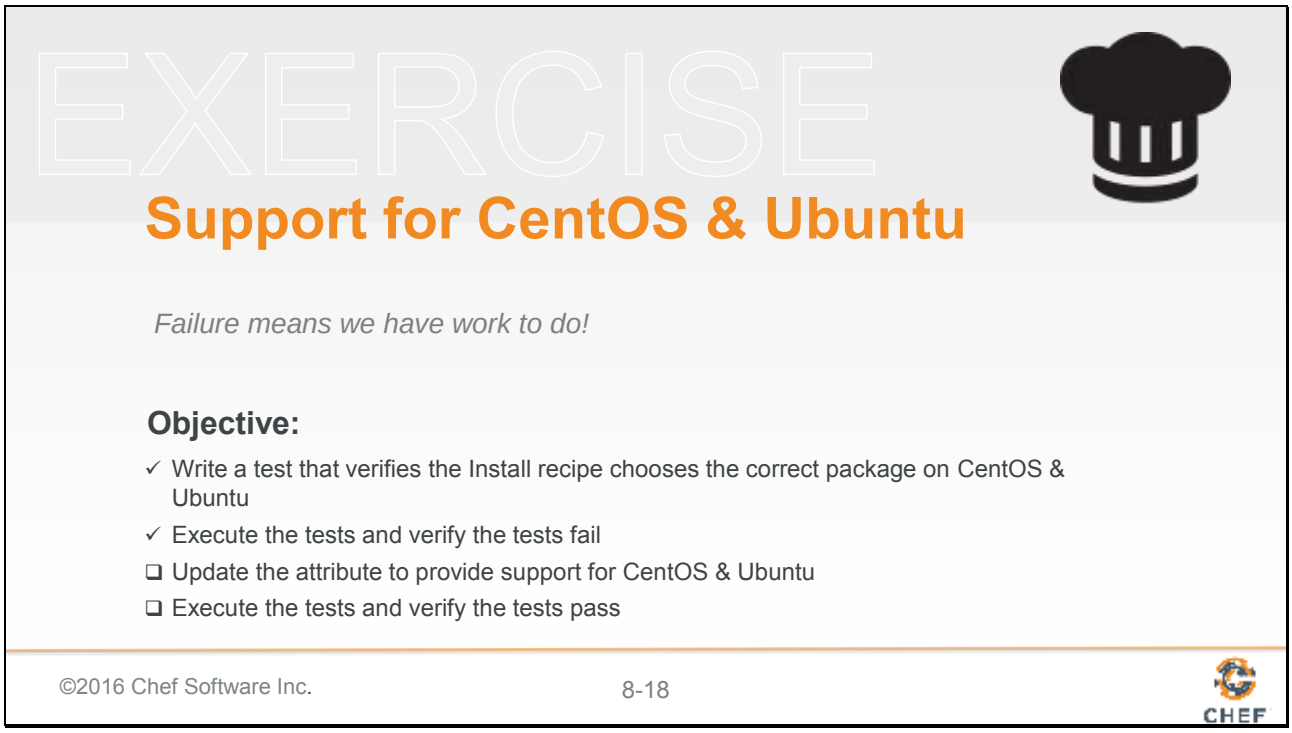


The examples have now been defined for the existing platform and the new platform.



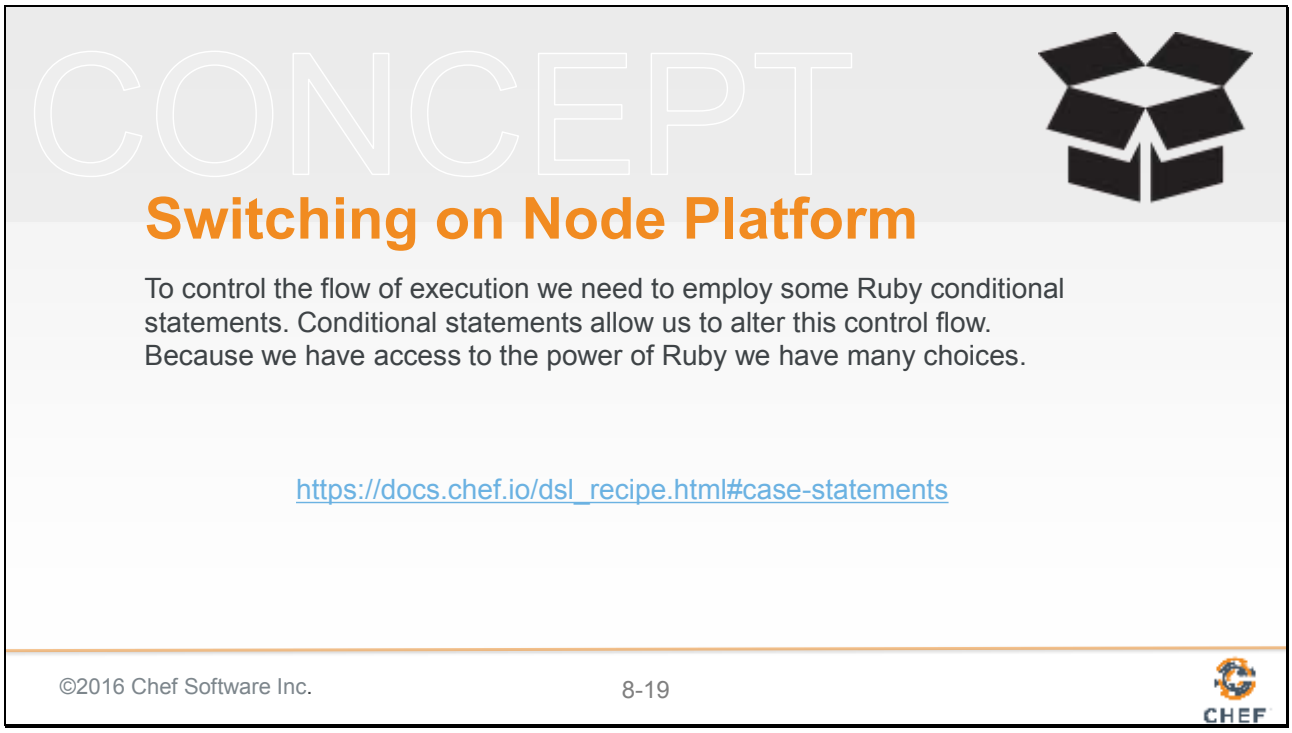
When it comes time to execute the tests we should see that defining the new platform will not raise an error when it converges but will fail to meet the expectation that we installed the correctly named package.

Slide 18

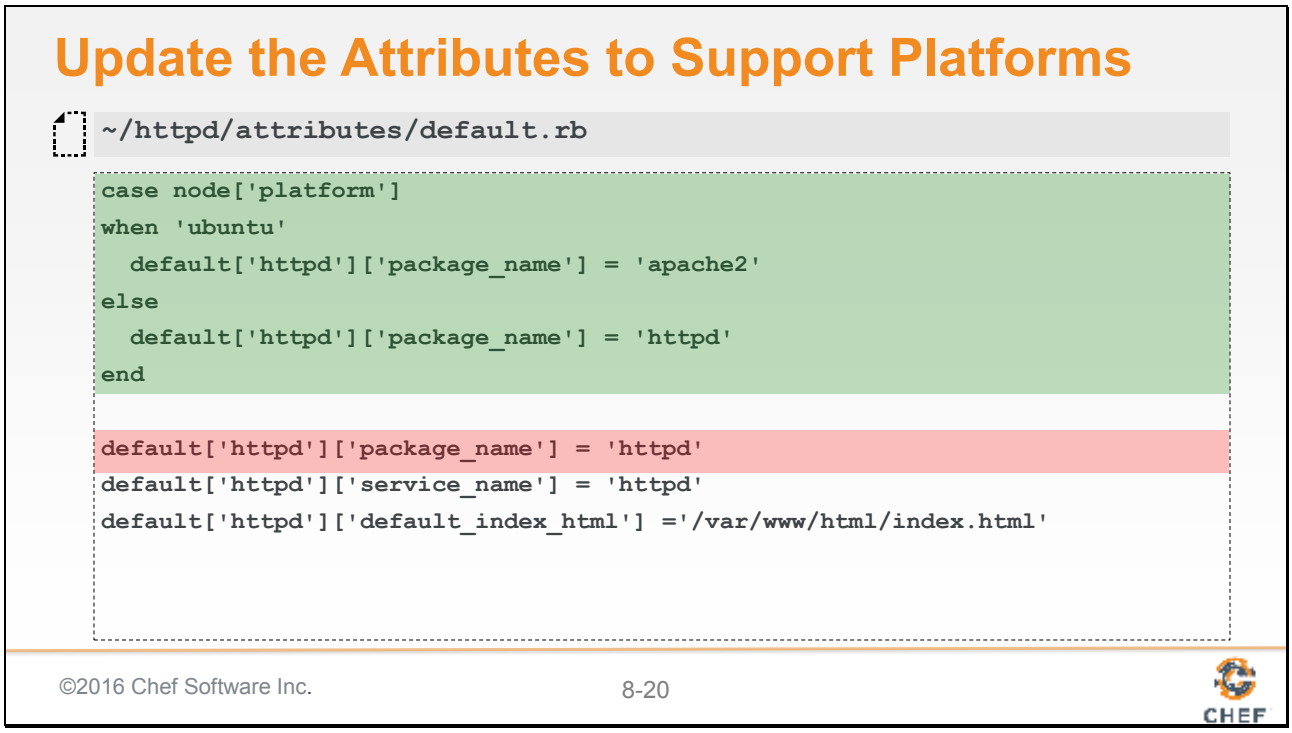


The name of the package is defined in the attributes file. That is what we refactored to support in the last section. It is now time to return to the attributes file and have it specify a different package name based on the platform.

Slide 19



To set the node attribute conditionally based on the platform means we are going to need to control the way that Ruby parses the code based on the state of the node platform. Ruby provides many ways to control the flow and several of them are documented in the recipe Domain Specific Language (DSL).



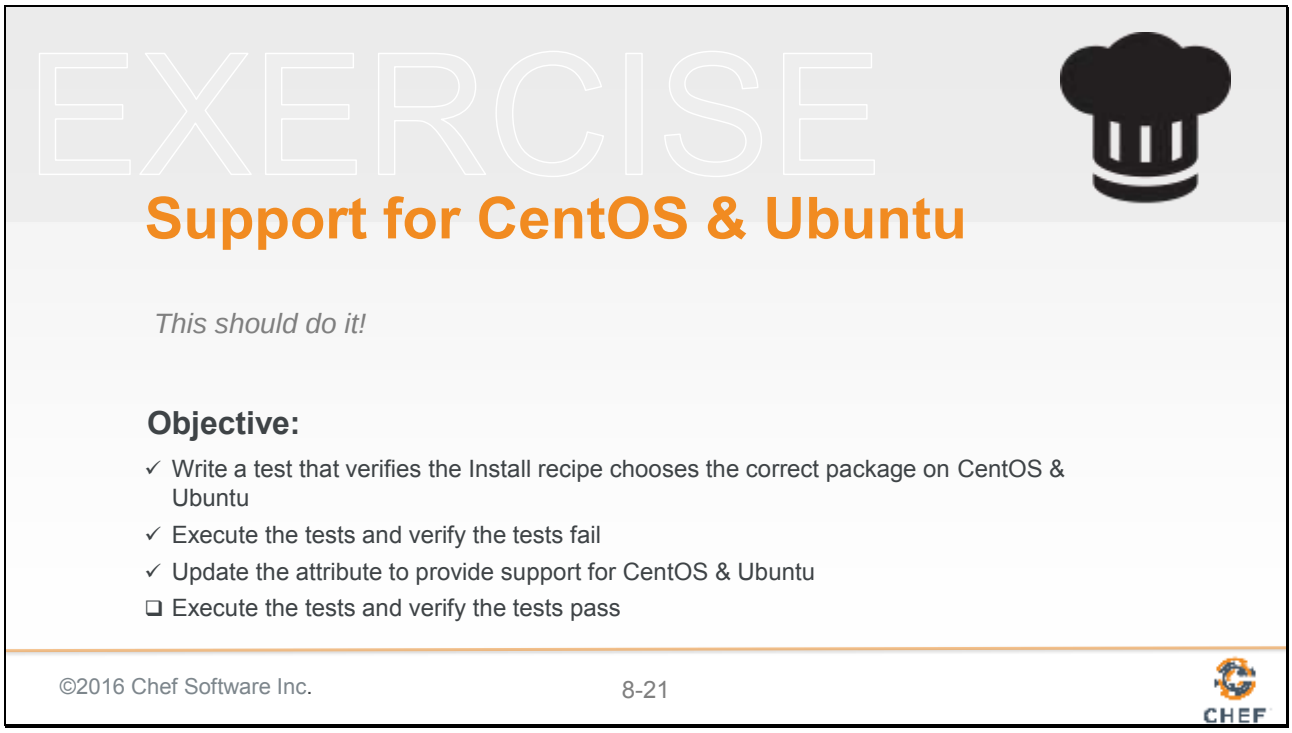
A very common way is to define a case statement. The case statement allows you to provide a value or value stored in a variable to the case keyword. Then following the case statement are a number of 'when' statements. Each 'when' needs to be provided with a value or value stored in a variable. If the value in the case statement equals the value in when statement then it is match and the flow of execution will take that path and ignore all others.

If none were to match we might be in trouble as the node attribute would never be set so we can use an 'else' statement which is as good as saying if none of those match then use this path.

The order of the case statement is particularly important as well. The first match that is made is the path the execution will take.

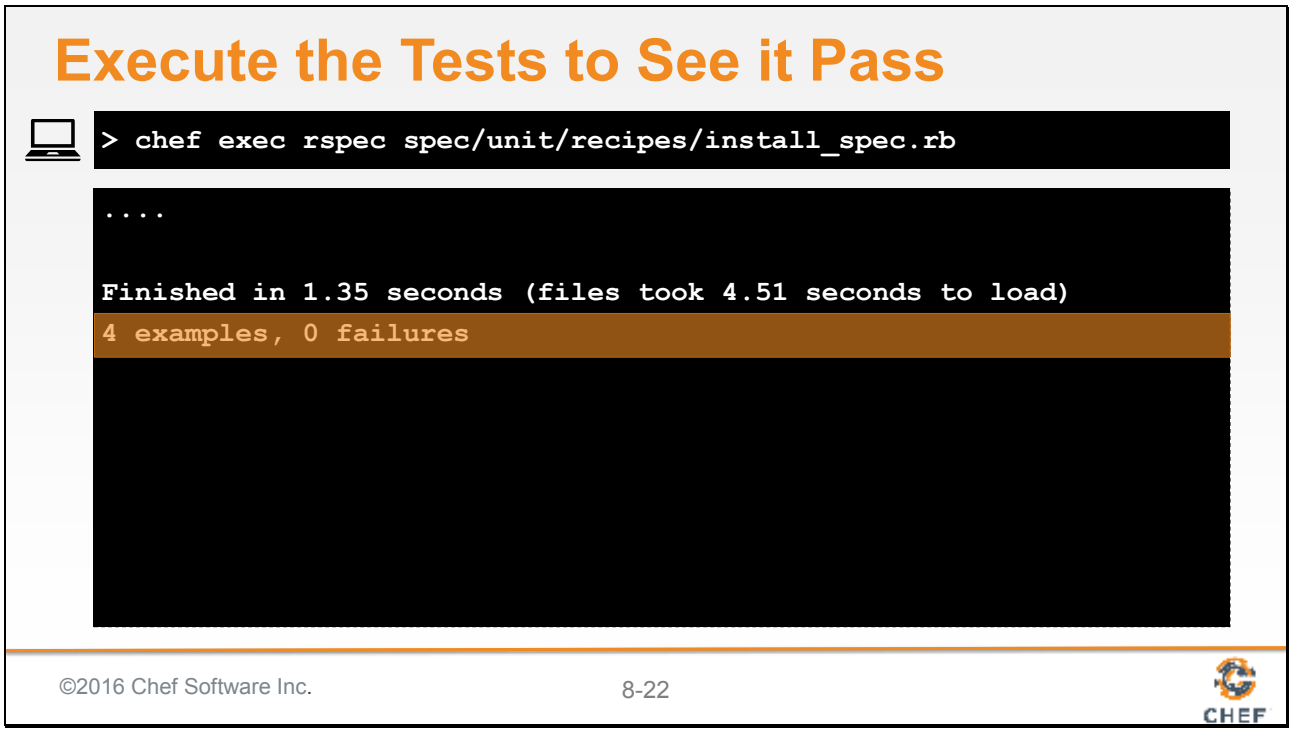
Instructor Note: When we say 'equal' each other we mean that Ruby is comparing the objects together with the equality method, the triple equals (===) .

Slide 21



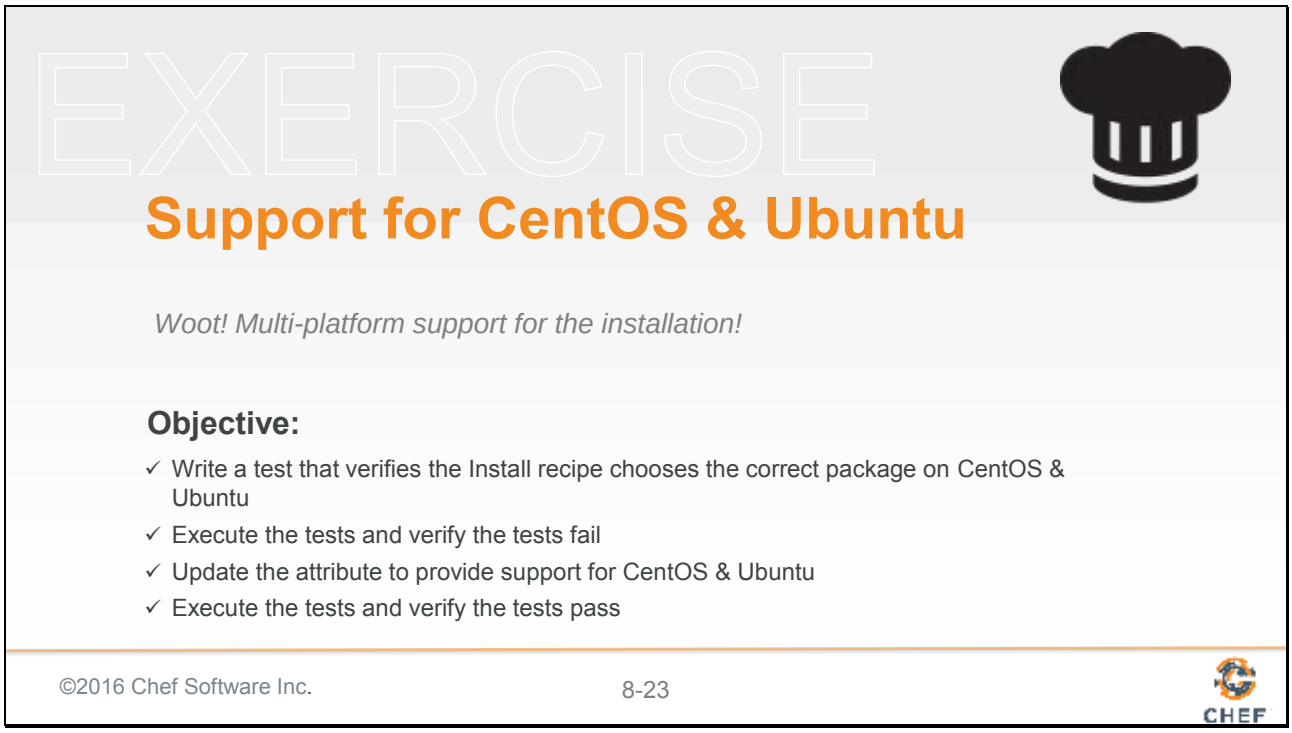
Now that the attributes file has been updated it is time execute the tests again and see if we defined this conditional logic correctly.

Slide 22



Executing the tests we should see both platforms will converge without error and install the necessary packages.

Slide 23



This approach to leverage the existing examples and use them to help define new examples for a new platform allowed us to build confidence through testing from the inside-out.

Taking this inside-out approach can feel right in situations where you know the steps you have to take.

Slide 24

LAB

Support for CentOS & Ubuntu

- Write a test that verifies the Service recipe chooses the service named 'httpd' on CentOS and 'apache2' on Ubuntu
- Execute the tests and verify the tests **fail**
- Update the attribute to choose the service name 'httpd' on CentOS and 'apache2' on Ubuntu
- Execute the tests and verify the tests **pass**

CHEF

Now as an exercise for you it is time to do the same thing for the service recipe. The service for Ubuntu is named 'apache2'. Start with the changes to the specifications, move through see the failure, update to use the same conditional statement structure and then see the examples verify your work.

Instructor Note: Allow 10 minutes to complete this exercise

Slide 25

Update the Context to be Platform Specific

~/spec/unit/recipes/service_spec.rb

```
describe 'httpd::service' do
  context 'When all attributes are default, on CentOS' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.7')
      runner.converge(described_recipe)
    end
    # ... it converges successfully ...

    it 'starts the appropriate service' do
      expect(chef_run).to start_service('httpd')
    end

    it 'enables the appropriate service' do
      expect(chef_run).to enable_service('httpd')
    end
    # ... SPECIFICATION CONTINUES ON THE NEXT SLIDE ...
```


Let's review the changes to the service specification. You start with ensuring the existing CentOS platform is explicitly stated in the context and defined in the parameters provided to the ServerRunner initialization.

Add a Second Context for Another Platform

~/spec/unit/recipes/service_spec.rb

```
  # ... CONTINUED FROM THE PREVIOUS SLIDE ...
  context 'When all attributes are default, on Ubuntu' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '14.04')
      runner.converge(described_recipe)
    end
    # ... it converges successfully ...

    it 'starts the appropriate service' do
      expect(chef_run).to start_service('apache2')
    end
    it 'enables the appropriate service' do
      expect(chef_run).to enable_service('apache2')
    end
  end
end
```


CHEF

You now define an entire example group dedicated to the Ubuntu platform which defines the same structure of examples but with the values that are important for the platform.

Executing the test you would see the appropriate failures for the correctly named services not being started and enabled.

Slide 28

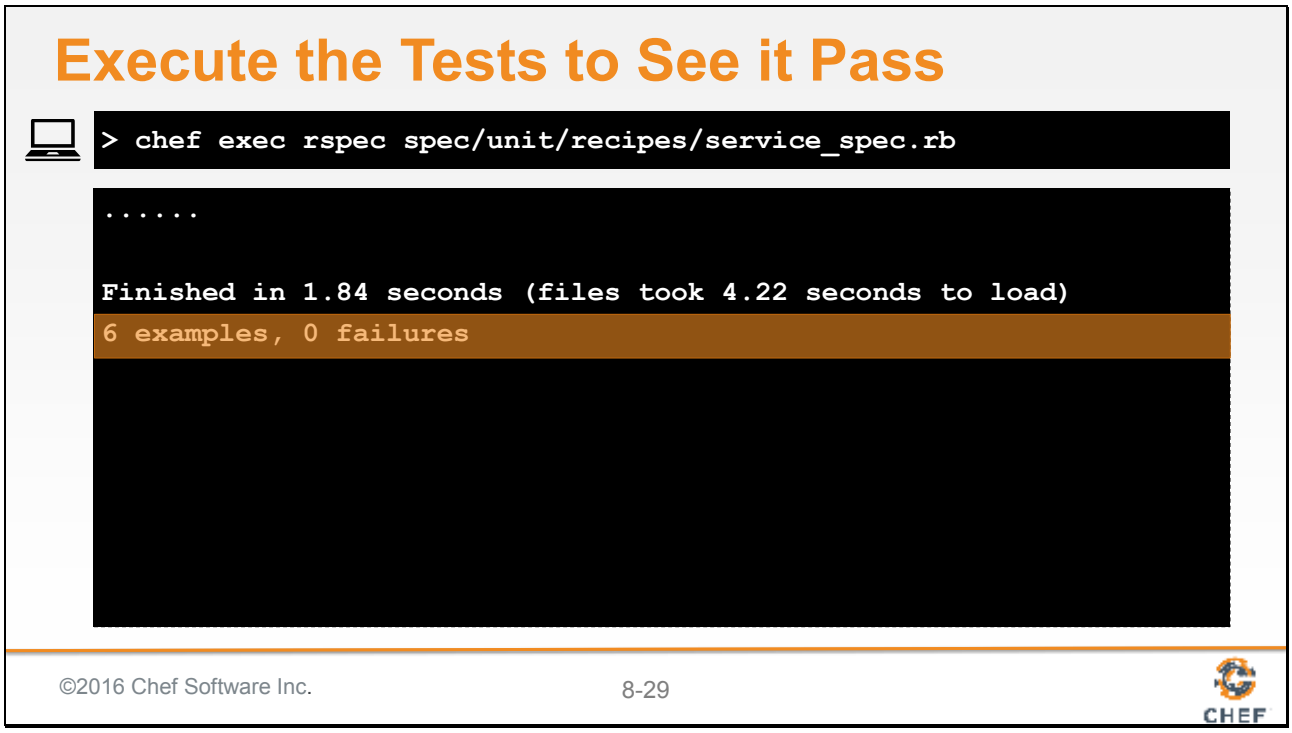
Update the Attribute to Support Platforms

~/httpd/attributes/default.rb

```
case node['platform']
when 'ubuntu'
  default['httpd']['package_name'] = 'apache2'
  default['httpd']['service_name'] = 'apache2'
else
  default['httpd']['package_name'] = 'httpd'
  default['httpd']['service_name'] = 'httpd'
end

default['httpd']['service_name'] = 'httpd'
default['httpd']['default_index_html'] = '/var/www/html/index.html'
```


Updating the attributes for the service should be a little less work because the structure is all in place.

Slide 29



Finally when we execute the tests again we see that all the examples pass.

Slide 30

LAB

Support for CentOS & Ubuntu

- ✓ Write a test that verifies the Service recipe chooses the service named 'httpd' on CentOS and 'apache2' on Ubuntu
- ✓ Execute the tests and verify the tests **fail**
- ✓ Update the attribute to choose the service name 'httpd' on CentOS and 'apache2' on Ubuntu
- ✓ Execute the tests and verify the tests **pass**

CHEF

That was nearly identical and a good way to reinforce the testing flow.

Slide 31

LAB

Support for CentOS & Ubuntu

- Write a test that verifies the file recipe chooses the same path (name) '/var/www/html/index.html' on CentOS and on Ubuntu
- Execute the tests that verify the tests **pass**
- Update the attribute to choose the same path on CentOS and on Ubuntu
- Execute the tests that verify the tests **pass**
- Get nervous! Mutate the attributes file!
- Undo the entire attributes change and verify the tests **pass**

Now only the configuration recipe remains. The default index HTML page for Ubuntu and CentOS are exactly the same. So when you define the new examples you actually will not see the failure. Then when you make the changes to the attributes file you will not see the failure. At that point you have written two new examples for the Ubuntu platform and it is important to ensure those tests fail. So pick a mutation (e.g. remove a line or specify an incorrect value) for the Ubuntu flow and ensure you see the failure.

Finally take a look at the code that you have created and ask yourself is that change better?

Instructor Note: Allow 10 minutes to complete this exercise

Slide 32

Update the Context to be Platform Specific

~/spec/unit/recipes/configuration_spec.rb

```
describe 'httpd::configuration' do
  context 'When all attributes are default, on CentOS' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'centos', version: '6.7')
      runner.converge(described_recipe)
    end
    # ... it converges successfully ...

    it 'creates a default index html page' do
      expect(chef_run).to create_file('/var/www/html/index.html')
    end

# ... SPECIFICATION CONTINUES ON THE NEXT SLIDE ...
```


Same as before we start with some maintenance of CentOS examples.

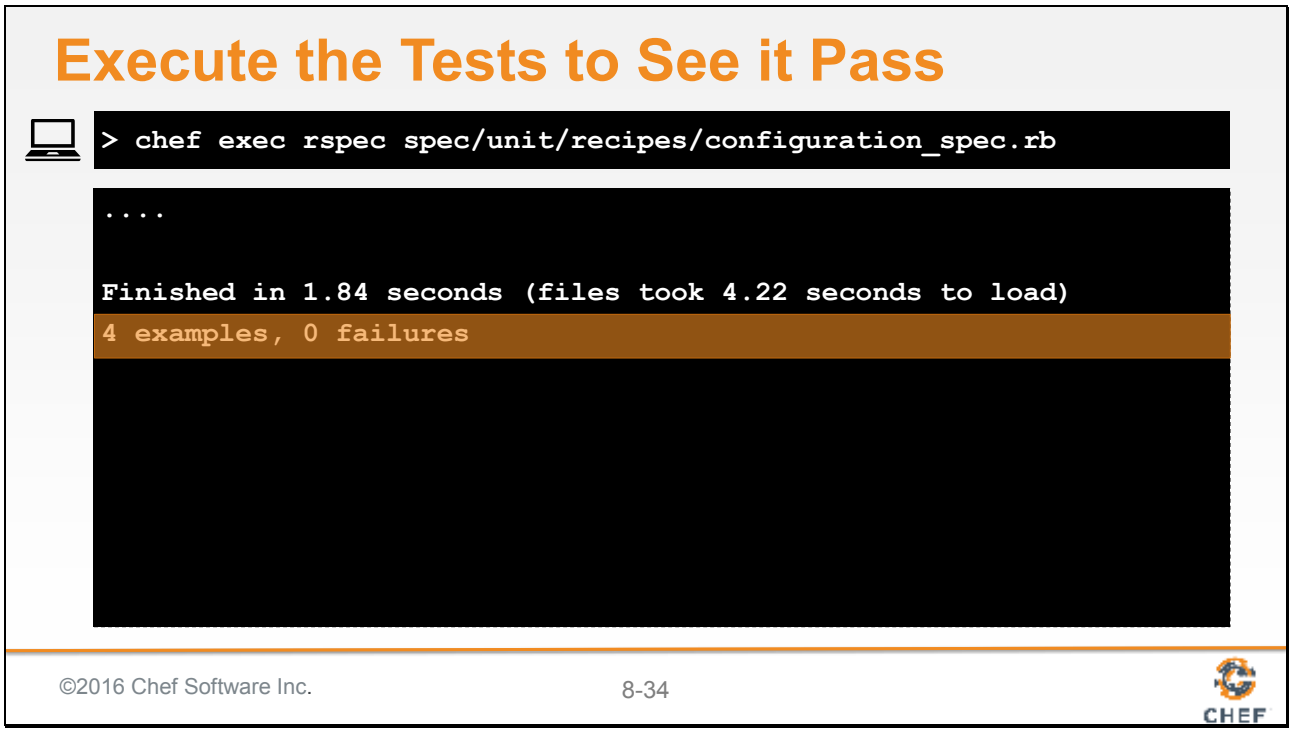
Slide 33

Add a Second Context for Another Platform

~/spec/unit/recipes/configuration_spec.rb

```
  # ... CONTINUED FROM THE PREVIOUS SLIDE ...
  context 'When all attributes are default, on Ubuntu' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new(platform: 'ubuntu', version: '14.04')
      runner.converge(described_recipe)
    end
    # ... it converges successfully ...

    it 'creates a default index html page' do
      expect(chef_run).to create_file('/var/www/html/index.html')
    end
  end
end
```


You then define another example group dedicated to the Ubuntu platform. Except this time the expectation is exactly the same.



Executing the tests shows you that everything is working.

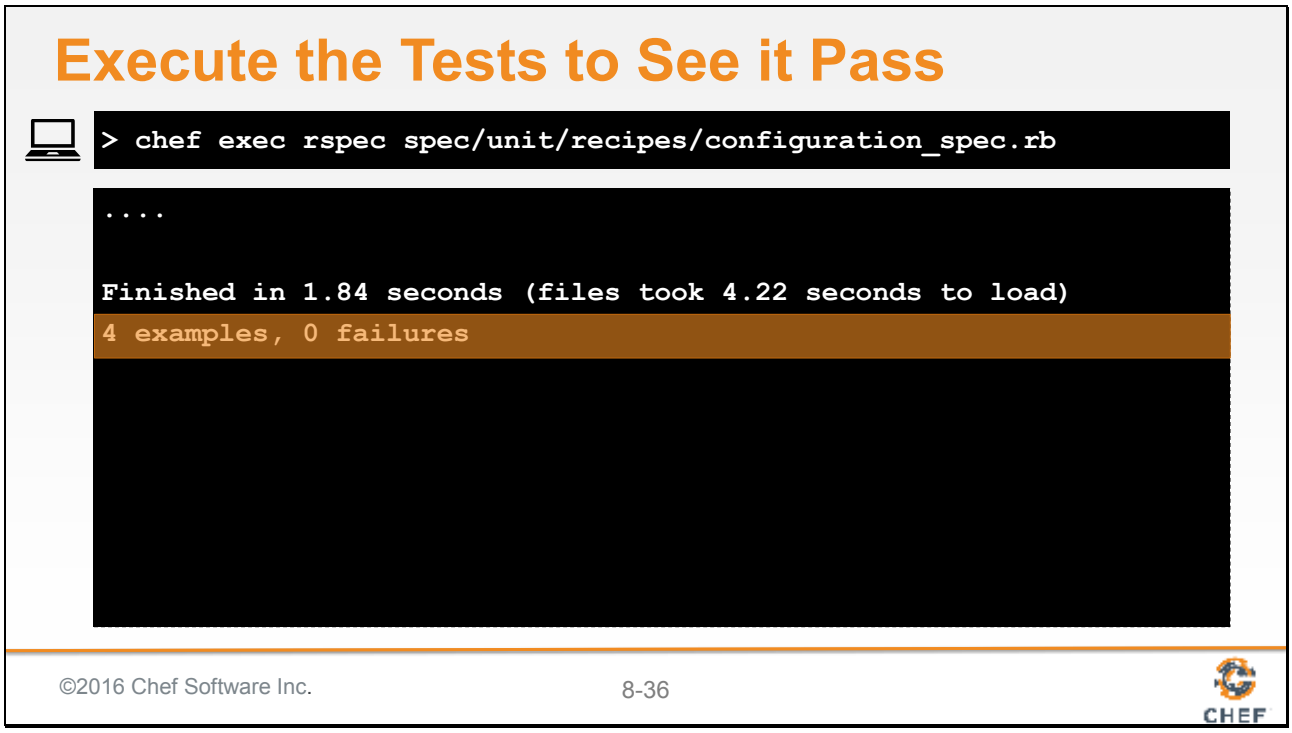
Update the Attribute to Support Platforms

~/httpd/attributes/default.rb

```
case node['platform']
when 'ubuntu'
  default['httpd']['package_name'] = 'apache2'
  default['httpd']['service_name'] = 'apache2'
  default['httpd']['default_index_html'] = '/var/www/html/index.html'
else
  default['httpd']['package_name'] = 'httpd'
  default['httpd']['service_name'] = 'httpd'
  default['httpd']['default_index_html'] = '/var/www/html/index.html'
end

default['httpd']['default_index_html'] = '/var/www/html/index.html'
```


CHEF

You implemented the change that we have done before.



And finally see the tests pass again. This is where you should become uncomfortable that we may have a false positive and that is a good time to ensure that you do not by mutating the code.

Slide 37

Update the Attribute to Support Platforms

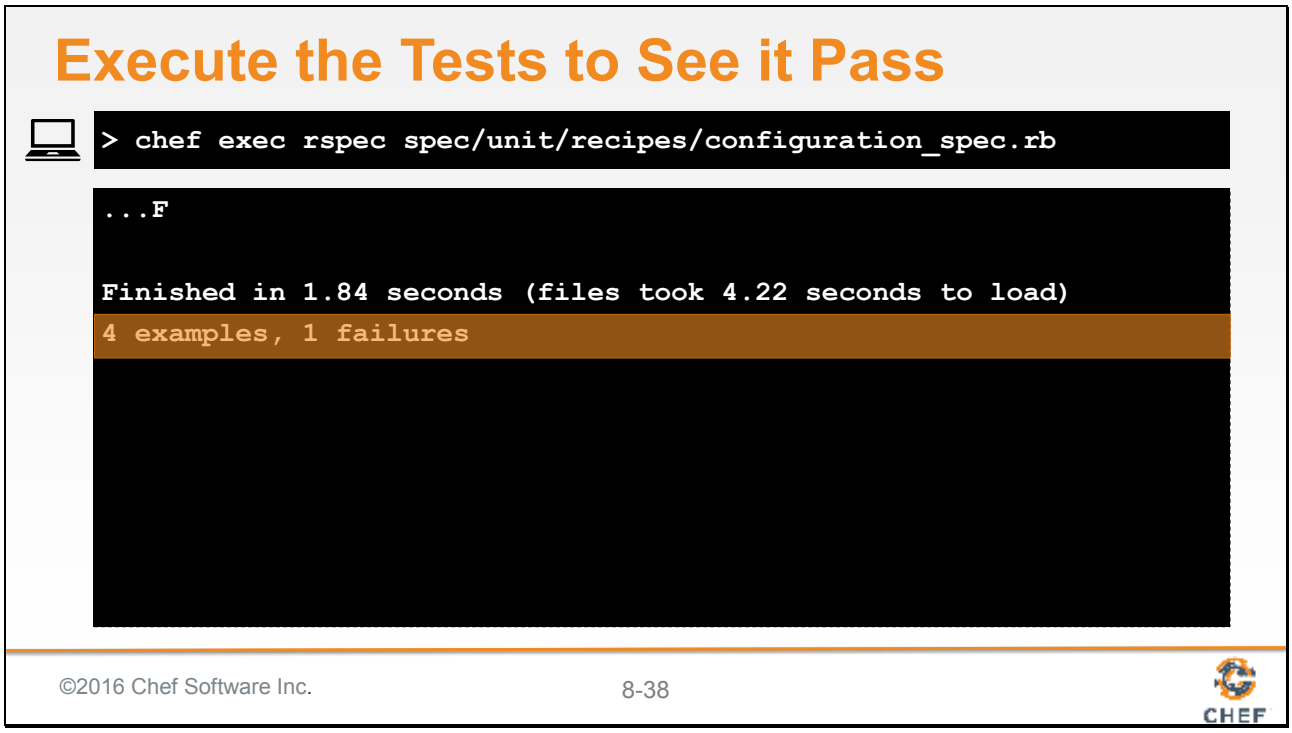
~/httpd/attributes/default.rb

```
case node['platform']
when 'ubuntu'
  default['httpd']['package_name'] = 'apache2'
  default['httpd']['service_name'] = 'apache2'
  default['httpd']['default_index_html'] = '/var/www/html/index.html2'
else
  default['httpd']['package_name'] = 'httpd'
  default['httpd']['service_name'] = 'httpd'
  default['httpd']['default_index_html'] = '/var/www/html/index.html'
end
```


CHEF

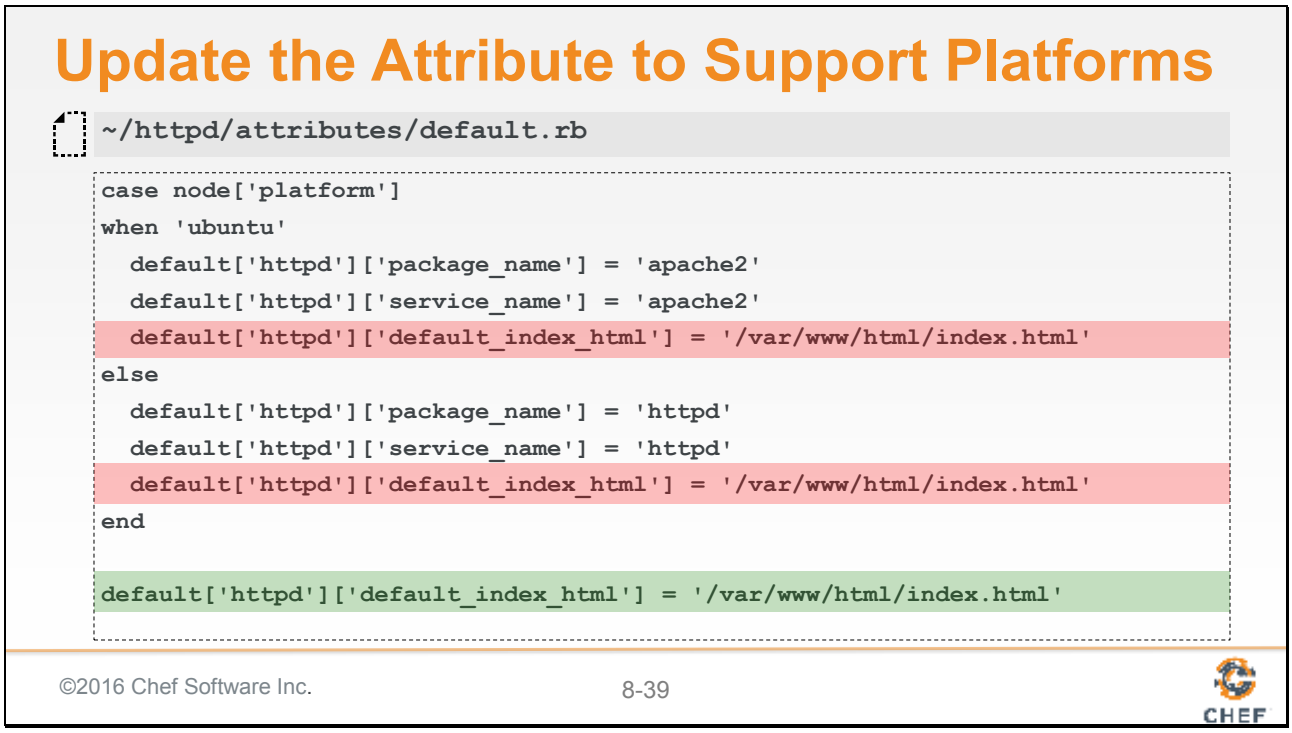
So anywhere in the Ubuntu flow of execution make a small mutation. In the example I am providing I have chosen a different path. Removing the attribute is another option as well.

Slide 38



Executing the tests should net at least one failure and that should give you more confidence that the expectations you have written are doing the work you want them to do.

Slide 39



Finally you might restore the code. Removing the mutation.

You may even choose to undo the change the proposed change. This is up to you to make the decision. In the example shown here I have returned to the original implementation. The original implementation worked, executing our tests proved it. Whether you should leave the attribute defined in the case statement or outside of it is up to you.

Leaving it in the case statements ensures that all values are defined on the platform. If a value on a particular platform were to change we would simply need to only change it within that platform's flow of control. However, if you never implement another platform you have created two lines of code. Some may argue the fewer lines of code you issue or statements you place inside of a conditional make it easier to read and understand.

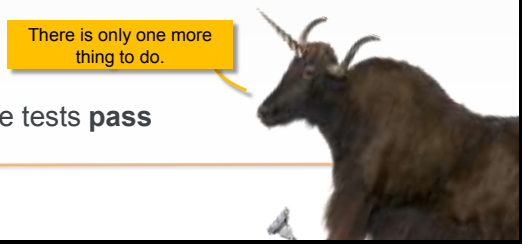
The most important thing is that the examples you defined should remain in the specification regardless of the implementation. The examples describe the expected behavior of the platform.

Slide 40

LAB

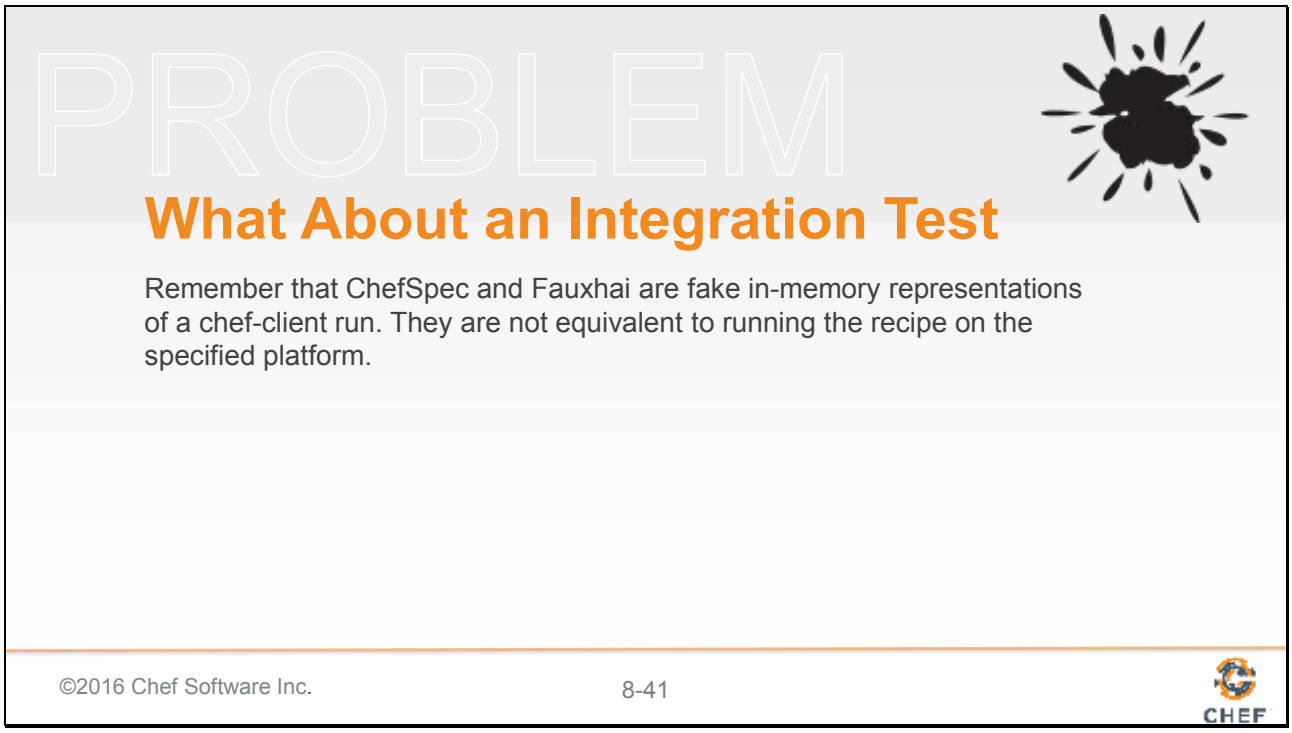
Support for CentOS & Ubuntu

- ✓ Write a test that verifies the file recipe chooses the same path (name) '/var/www/html/index.html' on CentOS and on Ubuntu
- ✓ Execute the tests that verify the tests **pass**
- ✓ Update the attribute to choose the same path on CentOS and on Ubuntu
- ✓ Execute the tests that verify the tests **pass**
- ✓ Get nervous! Mutate the attributes file!
- ✓ Undo the entire attributes change and verify the tests **pass**



Congratulations!

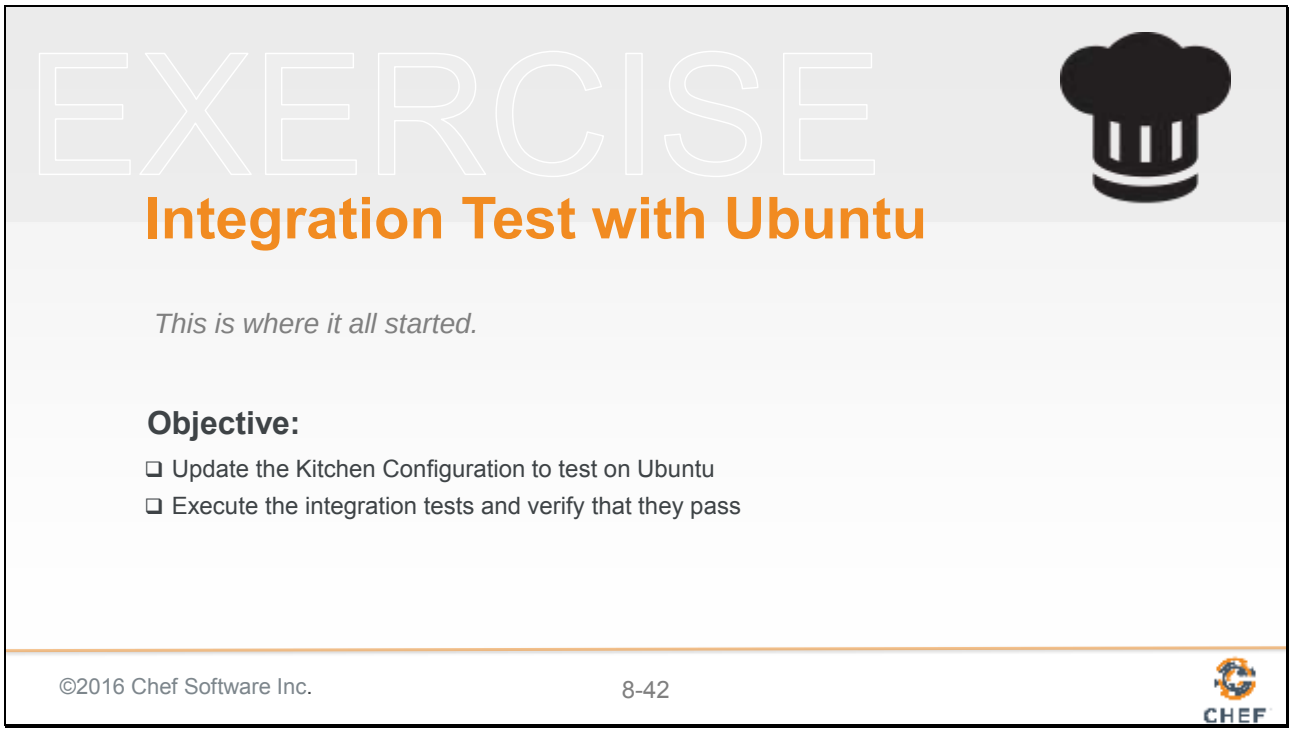
Slide 41



Now that we have finished building everything from the inside-out. It is finally time to see if the integration test works. This is important. When building recipes with ChefSpec you can very quickly make mistakes. Those mistakes are not the typos or omissions we have made. These are the mistakes that only the platform can catch.

Because we have been doing everything in-memory we really do not know if the package name, file path, or service name actually works. The only way to prove that is to apply the recipe to that platform.

Slide 42



So for our last and final exercise together lets update the Kitchen configuration to give us the ability to test on the Ubuntu platform.

Slide 43

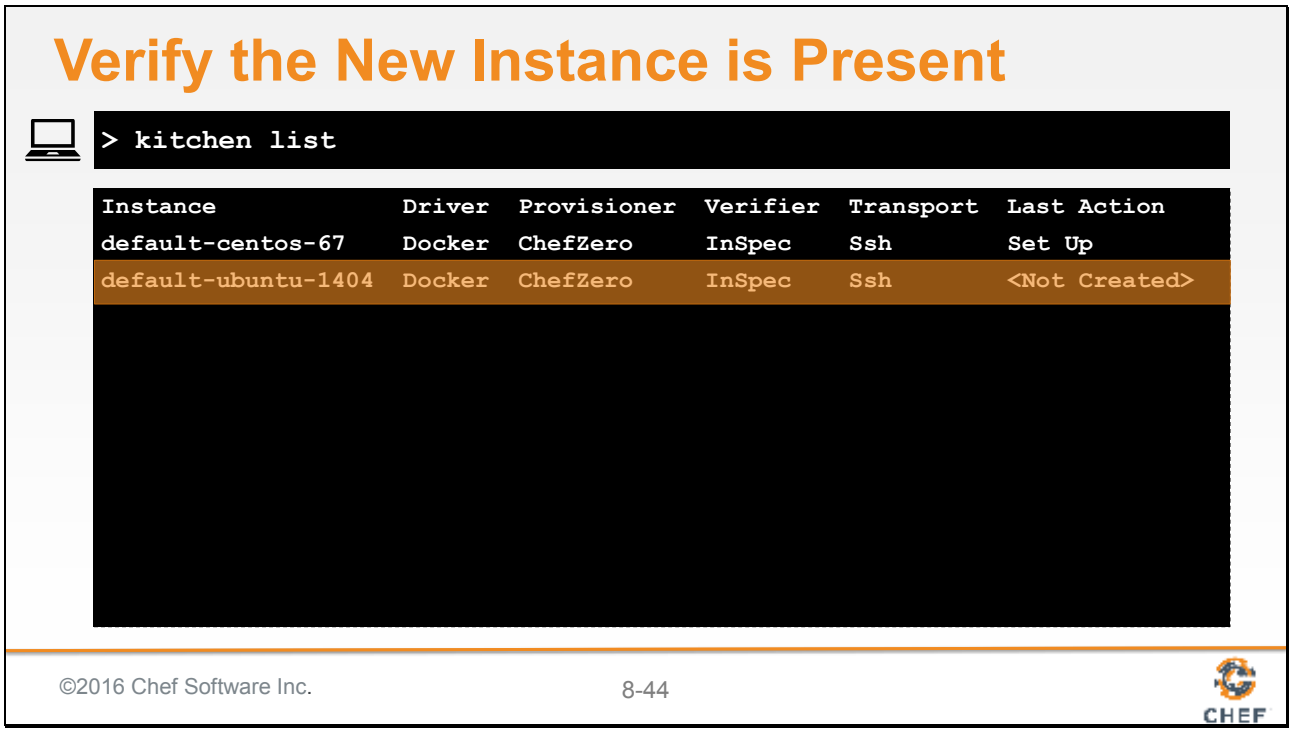
Add a New Platform to the Kitchen Configuration

~/httpd/.kitchen.yml

```
---
driver:
  name: docker

provisioner:
  name: chef_zero

verifier:
  name: inspec

platforms:
  - name: centos-6.7
  - name: ubuntu-14.04
```

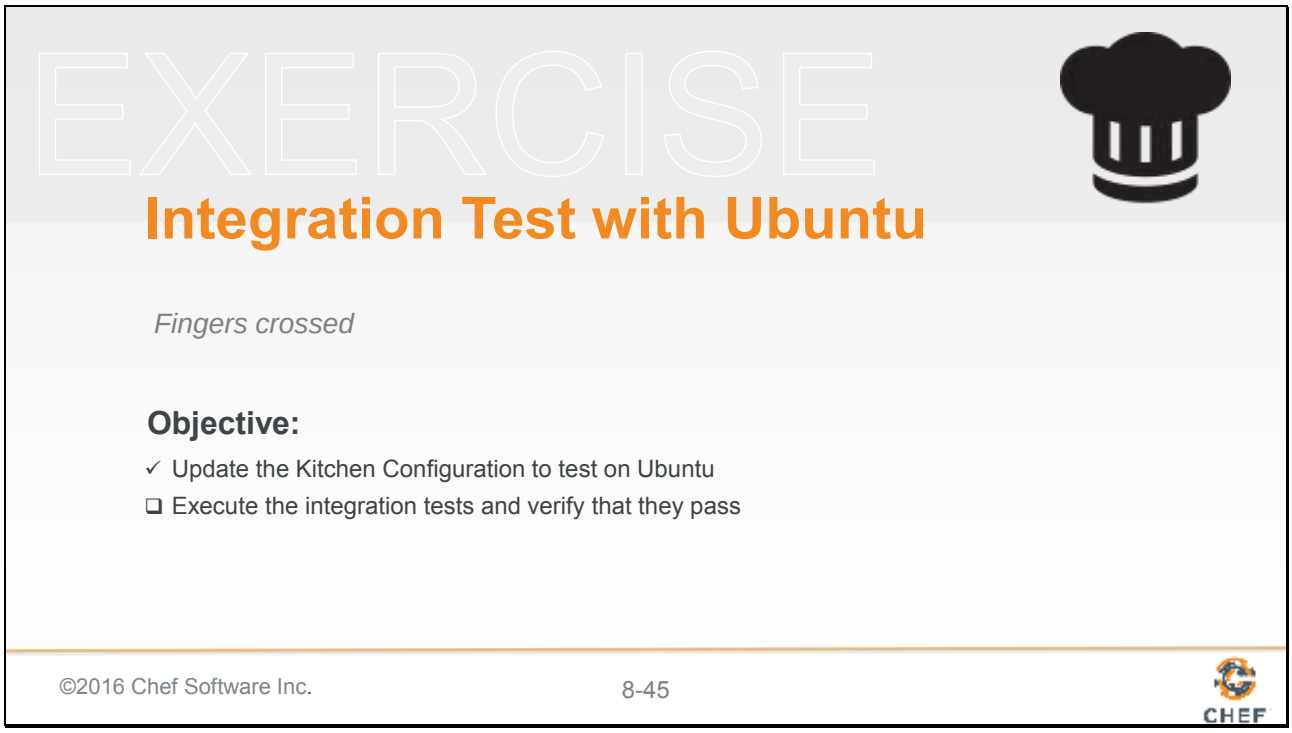

Within the kitchen configuration we define the new Ubuntu 14.04 platform.

Slide 44



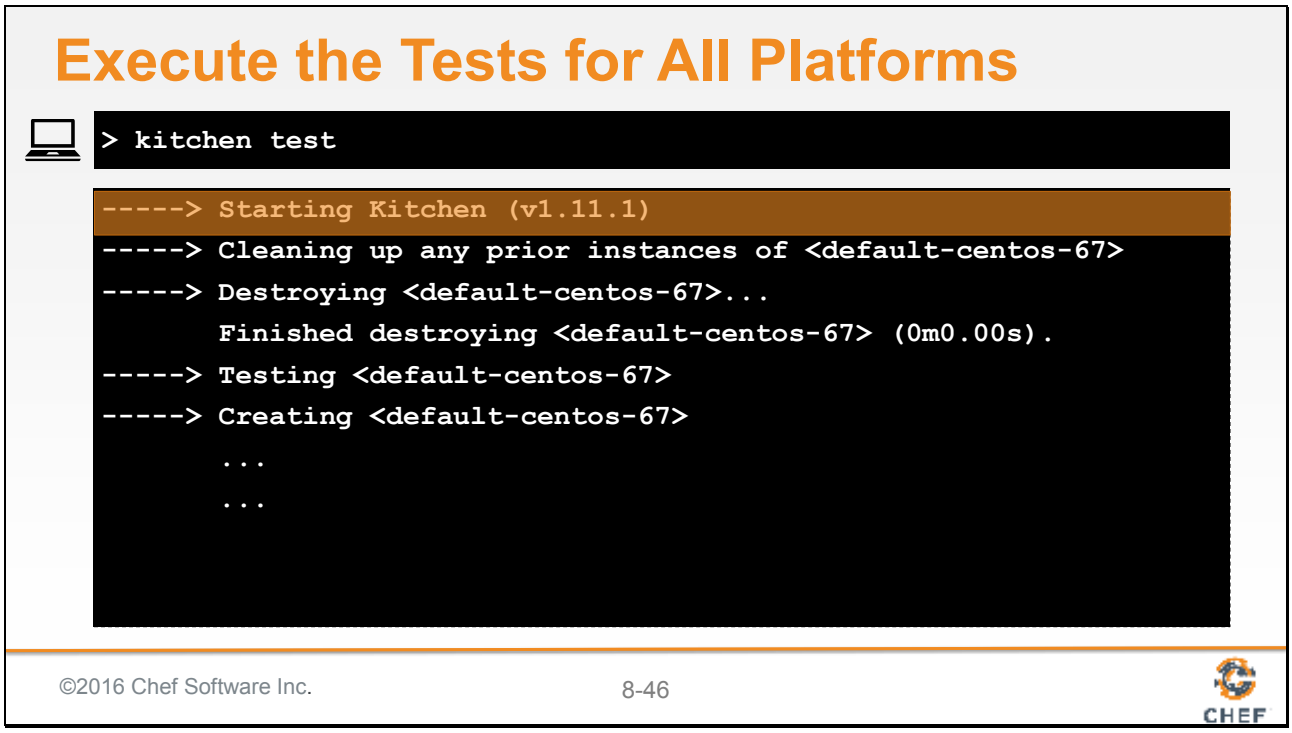
We verify that the platform exists within the list of instances.

Slide 45



And now it is time to execute the test suite. By choosing a very valuable and implementation free InSpec example, is the website up and running in localhost, we can be fairly certain that the expectations should be met.

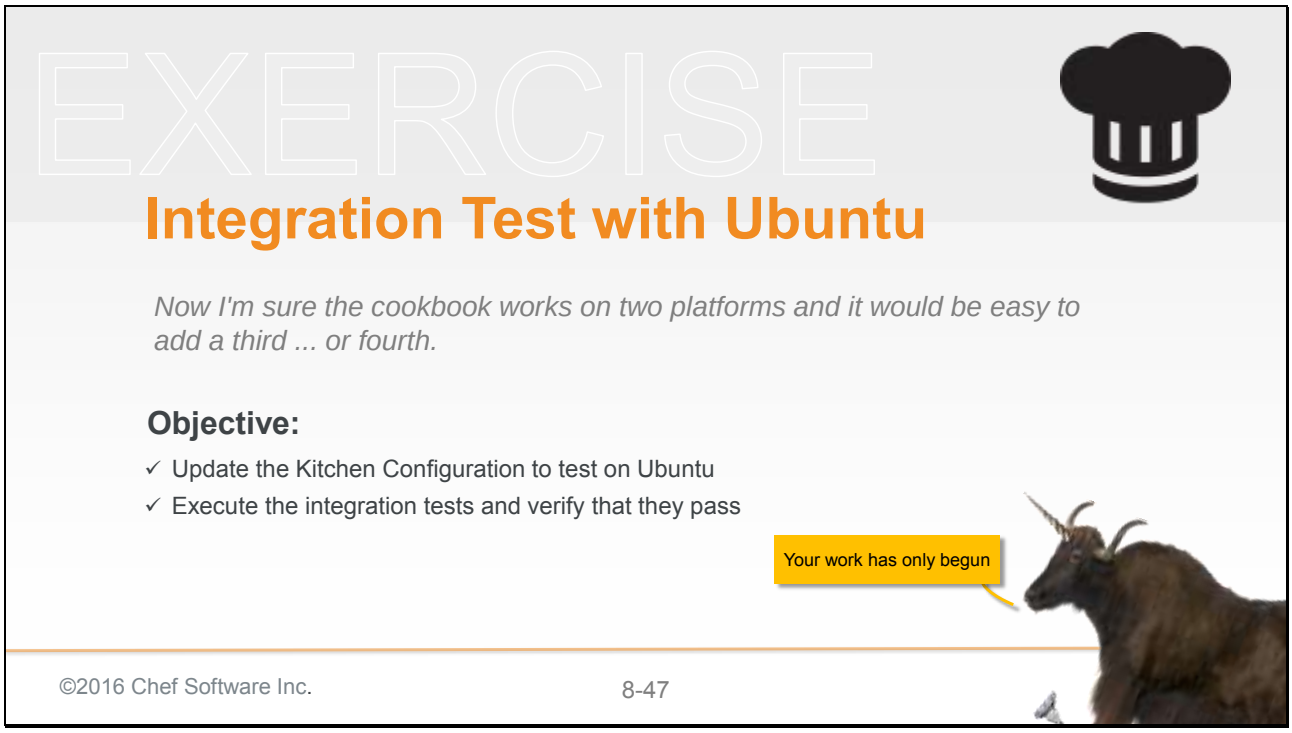
Slide 46



To execute the tests against both platforms run 'kitchen test'. Because we have two instances and did not specify a particular instance with the command it will run tests against all the listed instances.

This might be a good time to get up and move around as it will take some time.

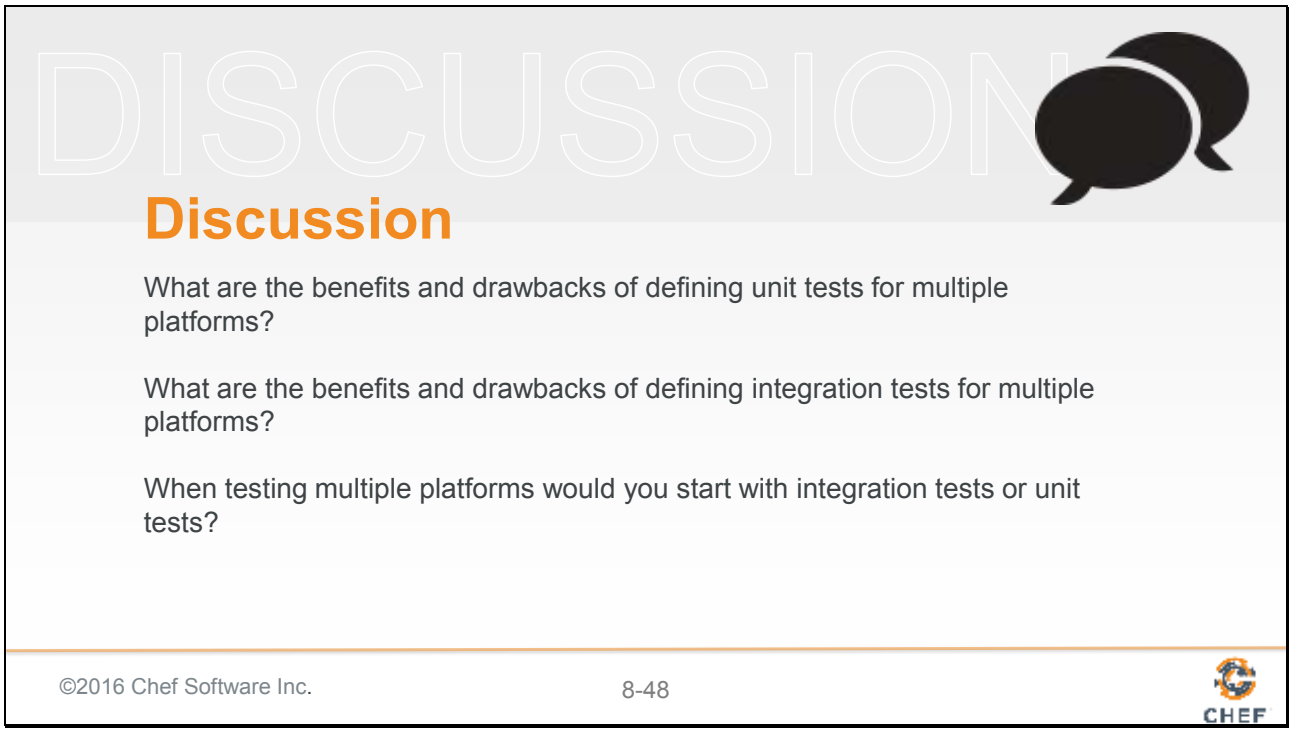
Slide 47



The expectations should pass and this brings the last exercise to a close.

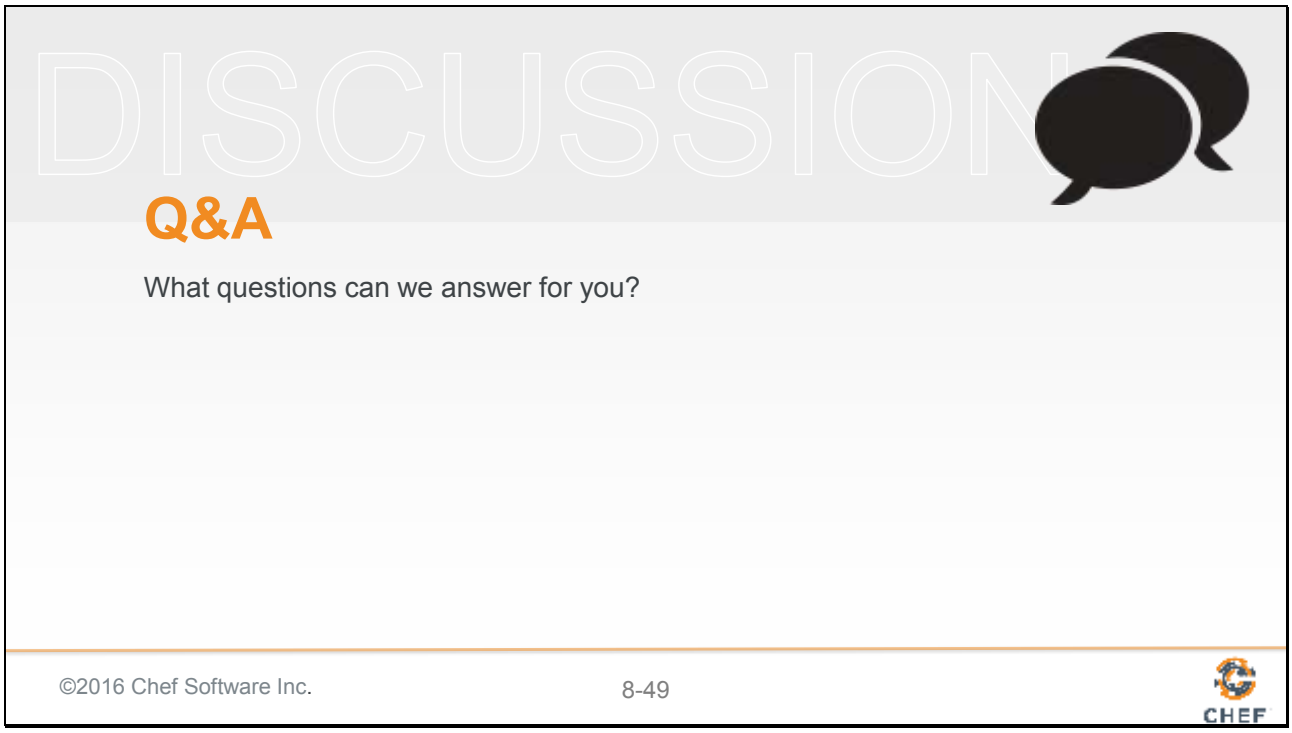
Let's have a discussion.

Slide 48



Instructor Note: With large groups I often find it better to have individuals turn to the individuals around them, form groups of whatever size they feel comfortable, and have them take turns asking and answering the questions. When all the groups are done I then open the discussion up to the entire group allowing each group or individuals to share their answers.

Slide 49



Before we complete this section, let us pause for questions.

Slide 50

Thank you for your time and attention.

Slide 51

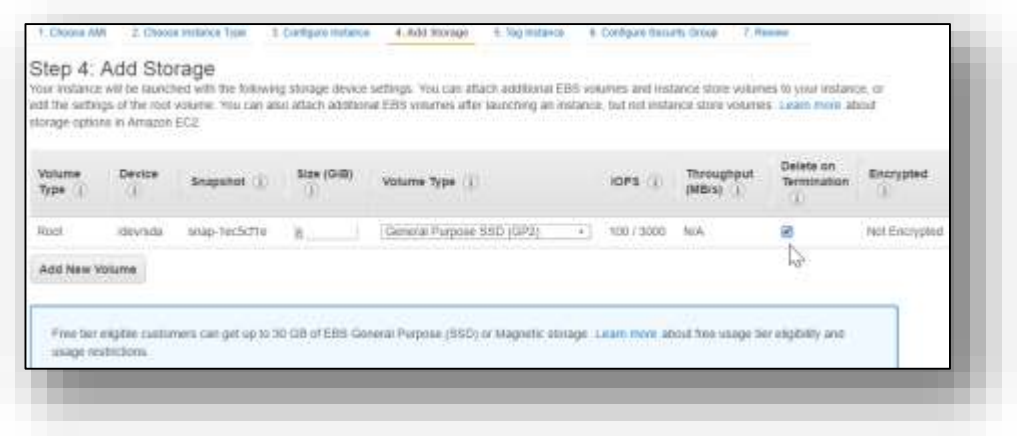
Appendix Z: Course-wide Instructor Notes

1. Training Lab System Setup

1. Open the AWS site from here: https://aws.amazon.com/
   - Login Credentials for Chef instructors: training-aws@chef.io
   - Password: Contact Chef Training Services if you don't know it or how to obtain it. training@chef.io
   - Partner credentials should be provided by Chef directly to partners.
2. Click the first link in column **EC2 *Virtual Servers in the Cloud***
3. From the navigation pane on the left, select **Images/AMIs**. The "Step 1" page displays with a list of available AMIs.
4. Select **TDD Cookbook Development – CentOS 6.7 – 1.1.0** from the list of options.
5. Click **Launch**. The "Step 2" page displays.
6. Select the first **Micro Instance** from the list provided and click **Next: Configure Instance Details** at the bottom of the screen. The "Step 3" page displays.
7. Enter the **Number of Instances**.

   Note: You will need 1 instance for each student enrolled in the class - and three for yourself.

8. Click **Next: Add Storage** at the bottom of the page. The "Step 4" page displays. Ensure that '**Delete on Termination**' is selected.



9. Click **Next: Tag Instance** at the bottom of the page. The "Step 5" page displays.
10. Enter a **Value**.

    Note: A recommended naming convention for the instances: [TRAINER'S INITIALS] - [CLASS NAME] - [CLASS DATE]

11. Click **Next: Configure Security Group**. The "Step 6" page displays.
12. Click the **Select an existing security group** radio button. A list of security groups displays.

13. Select **all-open**.
14. Click **Review and Launch** at the bottom of the screen. The "Step 7" page displays.
15. After you review the instances, click **Launch**. The "Select a key pair" window displays.
16. Confirm that this is set to **Proceed without a key pair** and click the acknowledgement check box.
17. Click **Launch Instances**. The "Launch Status" page displays.
18. Click **View Instances**. The instances list displays.
19. From here, copy all of the instances and create a gist file to share with the class.
20. Use goo.gl to shorten the URL to the gist file.

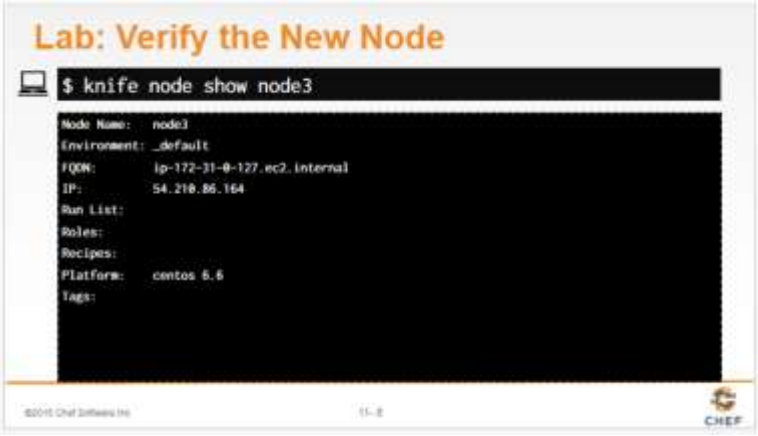
Note: The login credentials and password for the AMIs used in class are chef/chef. You'll need to tell the students that at the appropriate time.

2. How to Use Lab Slides

Regarding the "Lab" exercises (not the Group Exercises), you should encourage students to use the high-level hammer/wrench "Lab" slide steps first, and then resort to the subsequent detailed step slides if the students need the details to complete the lab. You can still use the subsequent detailed step slides as a vehicle to review each lab. For example:

This is a high-level hammer/wrench "Lab" instruction slide. Encourage students to complete the lab using this high level hammer/wrench "Lab" slide first.

If some students can't complete the lab based on the above slide, they are free to follow the subsequent detailed step slides, such as these:



You can also use the above detailed slides as a vehicle for reviewing the labs.